CLAIMS MANAGEMENT

Group Members:

Riju Chatterjee, Kalyani Government Engineering College, 009778 of 2019-2020

Ayan Ghorai, Kalyani Government Engineering College, 181020110016 of 2018-2019

Debjit Adhikary, Kalyani Government Engineering College, 181020110063 of 2018-2019

Subhodip Ghosh, Kalyani Government Engineering College, 181020110045 of 2018-2019

Shreya Mallick, Kalyani Government Engineering College, 201020101610006 of 2020-21

Table of Contents

Acknowledgement

I take this opportunity to express my profound gratitude and deep regards to my faculty Titas RoyChowdhury sir for his exemplary guidance, monitoring and constant encouragement throughout the course of this project. The blessing, help and guidance given by him time to time shall carry me a long way in the journey of life on which I am about to embark.

I am obliged to my project team members for the valuable information provided by them in their respective fields. I am grateful for their cooperation during the period of my assignment.

Riju Chatterjee

Acknowledgement

I take this opportunity to express my profound gratitude and deep regards to my faculty Titas RoyChowdhury sir for his exemplary guidance, monitoring and constant encouragement throughout the course of this project. The blessing, help and guidance given by him time to time shall carry me a long way in the journey of life on which I am about to embark.

I am obliged to my project team members for the valuable information provided by them in their respective fields. I am grateful for their cooperation during the period of my assignment.

Ayan Ghorai

Acknowledgement

I take this opportunity to express my profound gratitude and deep regards to my faculty Titas RoyChowdhury sir for his exemplary guidance, monitoring and constant encouragement throughout the course of this project. The blessing, help and guidance given by him time to time shall carry me a long way in the journey of life on which I am about to embark.

I am obliged to my project team members for the valuable information provided by them in their respective fields. I am grateful for their cooperation during the period of my assignment.

Debjit Adhikary

Acknowledgement

I take this opportunity to express my profound gratitude and deep regards to my faculty Titas RoyChowdhury sir for his exemplary guidance, monitoring and constant encouragement throughout the course of this project. The blessing, help and guidance given by him time to time shall carry me a long way in the journey of life on which I am about to embark.

I am obliged to my project team members for the valuable information provided by them in their respective fields. I am grateful for their cooperation during the period of my assignment.

Subhodip Ghosh

Acknowledgement

I take this opportunity to express my profound gratitude and deep regards to my faculty Titas RoyChowdhury sir for his exemplary guidance, monitoring and constant encouragement throughout the course of this project. The blessing, help and guidance given by him time to time shall carry me a long way in the journey of life on which I am about to embark.

I am obliged to my project team members for the valuable information provided by them in their respective fields. I am grateful for their cooperation during the period of my assignment.

Shreya Mallick

Project Objective

Machine learning for business:

The purpose of machine learning is to discover patterns in your data and then make predictions based on often complex patterns to answer business questions, detect and analyse trends and help solve problems.

Machine learning and predictive models can also equip insurers with a better understanding of claims costs. These insights can help a carrier save millions of dollars in claim costs through proactive management, fast settlement, targeted investigations and better case management.

Data has always played a central role in the insurance industry, and today, insurance carriers have access to more of it than ever before. We have created more data in the past two years than the human race has ever created. Insurers—like organisations in most industries—are overwhelmed by the explosion in data from a host of sources, including telematics, online and social media activity, voice analytics, connected sensors and wearable devices. They need machines to process this information and unearth analytical insights. But most insurers are struggling to maximise the benefits of machine learning. This situation is seeing a gradual but steady change, driven by an environment characterised by increased competition, elastic marketplaces, complex claims and fraud behaviour, higher customer expectations and tighter regulation. Insurers are being forced to explore ways to use predictive modelling and machine learning to maintain their competitive edge, boost business operations and enhance customer satisfaction. They are also examining how they can take advantage of recent advances in artificial intelligence (AI) and machine learning to solve business challenges across the insurance value chain. These include underwriting and loss prevention, product pricing, claims handling, fraud detection, sales and customer experience.

Problem Description:

We are provided with an anonymized dataset containing both categorical and numeric variables available when the claims were received by renowned insurance company. All string type variables are categorical. There are no ordinal variables.

The "target" column in the train set is the variable to predict. It is equal to 1 for claims suitable for an accelerated approval.

claims for which approval could be accelerated leading to faster payments ➔ 1
claims for which additional information is required before approval ➔ 0

Column’s names are encoded because of confidentiality.

What is objective:

Claim provision are crucial for the financial stability of insurance companies. Insurance industry is believing in Data Analytics to adopt the changing technologies to provide better facilities to their customers.

For that company future prediction of insurance claim will be very helpful to serve their customers a better service.

For an insurance company a person not eligible for claim(0), if he/she is approved for claim(1), this scenario is not intended. So we have to reduce the number of "False +ve" cases. Here the appropriate metrics are "Precision" and "F1_score".

So, we have to get best "Precision Score" and "F1 score" here and good accuracy should also be maintained.

Insurers are using machine learning to improve operational efficiency, from claims registration to claims settlement. Many carriers have already started to automate their claims processes, thereby enhancing the customer experience while reducing the claims settlement time. Machine learning and predictive models can also equip insurers with a better understanding of claims costs. These insights can help a carrier save millions of dollars in claim costs through proactive management, fast settlement, targeted investigations and better case management. Insurers can also be more confident about how much funding they allocate to claim reserves. Tokio Marine has an AI-assisted claim document recognition system that helps to handle handwritten claims notice documents using a cloud-based AI optical character recognition (OCR) service. It reduces 50 percent of the document input load as well as complies with privacy regulations. AI is used to read complicated, ambiguous Chinese characters (Kanji), and the "packetlike" data transfer system protects customer privacy. The results: over 90 percent recognition rate, 50 percent reduction in input time, 80 percent reduction in human error, and faster and hassle-free claims payments.

Abstract—Now a day's Data is playing a central role and is carrying the big asset in the insurance industry. In today's journey insurance industry has a vital role. Insurance transporters have access to more information than ever before. From the past 700+ years in the insurance industry we can consider the three major eras Starting from 15th century to 1960, industry followed the manual era, from1960s to 2000 we are in the systems era, now we are in digital era i.e. 2001-20X0.The highest corporate object in all three eras is that the fundamental insurance industry has been determined by believing the data analytics in adopting the changing technologies to better and keep the ways and keep capital together. In advanced analysis the main challenge is the analytical models and algorithms which are being insufficient to support insurers; only by machines we can overcome this challenge. Key Words — Machine Learning, Digital Era, Client Lifetime Value (CLV).

INTRODUCTION:

The most important advantage of Machine Learning (ML) to use in Insurance Industry is to facilitate data sets. Machine learning (ML) can be successfully useful across Structured, Semi Structured or Unstructured datasets. Machine learning can be used accurate across the value chain to identify with risk, claims and

customer actions, by means of advanced predictive accurateness. The probable applications of machine learning in insurance are plentiful from perceptive risk appetite and premium leakage, to expense administration, subrogation, proceedings and fraud detection. Machine learning is not a novel technology; this technology is following from the last few decades. There are 3 main categories of learning they are supervised learning, Unsupervised Learning and reinforcement learning. The greater part of the insurers are following Supervised Learning from last few decades for assessing the risk by means of known parameters in dissimilar combinations to acquire the preferred outcome. Present age insurers are motivated to unsupervised learning, in this predestined goals are clear. If there are any modifications in the variables, the method identifies those modifications and tries to change as per the goals. For example, according to traffic the GPS Suggests different routes dynamically based on traffic conditions. In insurance industry also the learning is adopted for usage based insurance. Reinforcement learning is mostly depends on ANN (Artificial Neuron Network), Target/ Goal can be modified dynamically depending on objective. Reinforcement learning is used for IOT applications.

Insurance Advice Machines will play a important role in client service, from organizing the preliminary communication to determine and cover a client necessities. According to a latest survey, a greater part of clients are satisfied to take delivery of computer-generated insurance recommendations. Clients are looking for tailored solutions which are made by machine learning algorithms that evaluate their profiles and recommend tailor-made products. PROPERTY/CASUALTY SMA Research, 2016 Innovation and Emerging Technologies. From the above figure n=84, Insurance business areas are machine learning and can be leveraged Machine learning is expansively used across the insurance value chain. One example is all state, which combined with (EIS) Early Information Science to develop a virtual assistant, called the Allstate Business Insurance Expert (ABIE).

The insurance companies are tremendously interested in the prediction of the future. Accurate prediction gives a probability to decrease financial loss for the company. The insurers use rather complex methodologies for this purpose. The major models are a decision tree, a random forest, a binary logistic regression, and a support vector machine. A great number of different variables are under analysis in this case. The algorithms involve detection of relations between claims, implementation of high dimensionality to reach all the levels, detection of the missing observations, etc. In this way, the individual customer's portfolio is made. Forecasting the upcoming claims helps to charge competitive premiums that are not too high and not too low. It also contributes to the improvement of the pricing models. This helps the insurance company to be one step ahead of its competitor.

How we are planning to solve:

In the Problem Statement of the dataset, it is clearly mentioned that we have to predict the target variable that claim for which approval can accelerate leading to faster payments or more additional information is required before approval.

That means, we have to categorize the target data into the certain group of ones or zeros.

So, it is undoubtedly a classification problem.

So, Here The classification problem is binomial type.

1. Data Description:

At first we load the data to our code here and we display the columns, some data from data set, observe the null value counts.

2. Exploratory Data Analysis:

Now we plot graph of each column w.r.t. target to see how the columns are effecting our target.

3. Data cleaning:

Here we drop some columns which are not effecting our target. And we handle Null values.

4. Data Pre-processing:

Here we apply pre-processing to normalize our data and by Recursive Feature Elimination and Variance Inflation Factor we reduce number of columns for final model.

5. Model Building:

It is the most important step in Machine Learning. We apply various models on our dataset and watch for best results. We have used the following models to train our dataset:

1. Logistic Regression
2. Decision Tree
3. Naive Bayes
4. Random Forest

Data Description

Source of data:

We collected the data of this project, from Kaggle's official site. In which the claims were received by a renowned insurance company. And they share their real dataset.

www.Kaggle.com/datasets

Each feature/column description:

The collected dataset is an anonymized dataset containing both categorical and numeric variables available when the claims were received by renowned insurance company. All string type variables are categorical. There are no ordinal variables.

Shape of the dataset:

```
print(df.shape)   ## Gives the total no of rows & columns
```

(114321, 133)

So there are 133 features(columns) and 114321 rows

Feature name:

Column names in the dataset are masked because of customer confidentiality.

The column names are like …

```
for col in df:
    print(col,end=" ")
```

ID target v1 v2 v3 v4 v5 v6 v7 v8 v9 v10 v11 v12 v13 v14 v15 v16 v17 v18 v19 v20 v21 v22 v23 v24 v25 v26 v27 v28 v29 v30 v31 v32 v33 v34 v35 v36 v37 v38 v39 v40 v41 v42 v43 v44 v45 v46 v47 v48 v49 v50 v51 v52 v53 v54 v55 v56 v57 v58 v59 v60 v61 v62 v63 v64 v65 v66 v67 v68 v69 v70 v71 v72 v73 v74 v75 v76 v77 v78 v79 v80 v81 v82 v83 v84 v85 v86 v87 v88 v89 v90 v91 v92 v93 v94 v95 v96 v97 v98 v99 v100 v101 v102 v103 v104 v105 v106 v107 v108 v109 v110 v111 v112 v113 v114 v115 v116 v117 v118 v119 v120 v121 v122 v123 v124 v125 v126 v127 v128 v129 v130 v131

Dataset full information:

```
df.info()  ## To get which columns are numeric
```

<class 'pandas.core.frame.DataFrame'>

RangeIndex: 114321 entries, 0 to 114320

Columns: 133 entries, ID to v131

dtypes: float64(108), int64(6), object(19)

memory usage: 116.0+ MB

So, there are 114 numeric columns and 19 Object column.

Total 133 columns.

From the 114 numeric columns, 108 are of float type and 6 are discrete.

Short view of dataset:

```
df.head()
```

	ID	target	v1	v2	v3	v4	v5	v6	v7	v8	...	v122	v123	v124	v125	v126	v127	v128	v129	v130	v131
0	3	1	1.335739	8.727474	C	3.921026	7.915266	2.599278	3.176895	0.012941	...	8.000000	1.989780	0.035754	AU	1.804126	3.113719	2.024285	0	0.636365	2.857144
1	4	1	NaN	NaN	C	NaN	9.191265	NaN	NaN	2.301630	...	NaN	NaN	0.598896	AF	NaN	NaN	1.957825	0	NaN	NaN
2	5	1	0.943877	5.310079	C	4.410969	5.326159	3.979592	3.928571	0.019645	...	9.333333	2.477596	0.013452	AE	1.773709	3.922193	1.120468	2	0.883118	1.176472
3	6	1	0.797415	8.304757	C	4.225930	11.627438	2.097700	1.987549	0.171947	...	7.018256	1.812795	0.002267	CJ	1.415230	2.954381	1.990847	1	1.677108	1.034483
4	8	1	NaN	NaN	C	NaN	NaN	NaN	NaN	NaN	...	NaN	NaN	NaN	Z	NaN	NaN	NaN	0	NaN	NaN

5 rows × 133 columns

Description of Object Columns:

```
df.describe(include="object")
```

	v3	v22	v24	v30	v31	v47	v52	v56	v66	v71	v74	v75	v79	v91	v107	v110	v112	v113	v125
count	110864	113821	114321	54211	110864	114321	114318	107439	114321	114321	114321	114321	114321	114318	114318	114321	113939	59017	114244
unique	3	18210	5	7	3	10	12	122	3	9	3	4	18	7	7	3	22	36	90
top	C	AGDF	E	C	A	C	J	BW	A	F	B	D	C	A	E	A	F	G	BM
freq	110584	2386	55177	32178	88347	55425	11103	11351	70353	75094	113560	75087	34561	27079	27079	55688	21671	16252	5759

In case of Categorical Object column , it includes the total value_count, number of unique values and most frequent data and its frequency.

Numeric Columns description with 5 number statistics:

The 5 number statistics is basically the minimum_value, maximum_value , 25% quantile, 50% quantile and 75% quantile.

And the full description includes the total_count, mean, standard deviation and 5 number statistics

For discrete columns:

```
df.describe(include="int64")
```

	ID	target	v38	v62	v72	v129
count	114321.000000	114321.000000	114321.000000	114321.000000	114321.000000	114321.000000
mean	114228.928228	0.761199	0.090928	1.030694	1.431767	0.310144
std	65934.487362	0.426353	0.583478	0.696244	0.922267	0.693262
min	3.000000	0.000000	0.000000	0.000000	0.000000	0.000000
25%	57280.000000	1.000000	0.000000	1.000000	1.000000	0.000000
50%	114189.000000	1.000000	0.000000	1.000000	1.000000	0.000000
75%	171206.000000	1.000000	0.000000	1.000000	2.000000	0.000000
max	228713.000000	1.000000	12.000000	7.000000	12.000000	11.000000

For Continues Columns:

```
df[floatcols[0:20]].describe()
```

	v1	v2	v4	v5	v6	v7	v8	v9	v10	v11	v12
count	6.448900e+04	6.452500e+04	6.452500e+04	6.569700e+04	6.448900e+04	6.448900e+04	6.570200e+04	6.447000e+04	1.142370e+05	6.448500e+04	1.142350e+05
mean	1.630686e+00	7.464411e+00	4.145098e+00	8.742359e+00	2.436402e+00	2.483921e+00	1.496569e+00	9.031859e+00	1.883046e+00	1.544741e+01	6.881304e+00
std	1.082813e+00	2.961676e+00	1.148263e+00	2.036018e+00	5.999653e-01	5.894485e-01	2.783003e+00	1.930262e+00	1.393979e+00	7.900790e-01	9.244945e-01
min	-9.996497e-07	-9.817614e-07	-6.475929e-07	-5.287068e-07	-9.055091e-07	-9.468765e-07	-7.783778e-07	-9.828757e-07	-9.875317e-07	-1.459062e-07	5.143224e-07
25%	9.135798e-01	5.316428e+00	3.487398e+00	7.605918e+00	2.065064e+00	2.101477e+00	8.658986e-02	7.853659e+00	1.050328e+00	1.500000e+01	6.322471e+00
50%	1.469550e+00	7.023803e+00	4.205991e+00	8.670867e+00	2.412790e+00	2.452166e+00	3.860317e-01	9.059582e+00	1.312910e+00	1.549595e+01	6.612969e+00
75%	2.136128e+00	9.465497e+00	4.833250e+00	9.771353e+00	2.775285e+00	2.834285e+00	1.625246e+00	1.023256e+01	2.100657e+00	1.595000e+01	7.019983e+00
max	2.000000e+01	2.000000e+01	2.000000e+01	2.000000e+01	2.000000e+01	2.000000e+01	2.000000e+01	2.000000e+01	1.853392e+01	2.000000e+01	1.871055e+01

v13	v14	v15	v16	v17	v18	v19	v20	v21
6.448900e+04	1.143170e+05	6.448500e+04	6.442600e+04	6.452500e+04	6.448900e+04	6.447800e+04	64481.000000	113710.000000
3.798396e+00	1.209428e+01	2.080911e+00	4.923222e+00	3.832270e+00	8.410455e-01	2.223005e-01	17.773592	7.029740
1.175892e+00	1.443947e+00	7.329166e-01	1.791187e+00	1.911504e+00	6.162762e-01	1.713459e-01	1.155002	1.072271
-8.464889e-07	-9.738831e-07	-8.830427e-07	-9.978294e-07	-9.066455e-07	4.475470e-07	-5.178987e-07	1.516776	0.106181
3.067998e+00	1.125602e+01	1.611419e+00	3.864735e+00	2.699122e+00	5.095767e-01	1.739154e-01	17.330703	6.415535
3.591081e+00	1.196783e+01	1.992031e+00	4.932127e+00	3.554267e+00	7.739063e-01	1.986554e-01	18.036585	7.045416
4.289486e+00	1.271577e+01	2.418676e+00	5.957448e+00	4.512221e+00	1.071453e+00	2.386316e-01	18.542631	7.670577
2.000000e+01	2.000000e+01	2.000000e+01	2.000000e+01	2.000000e+01	2.000000e+01	2.000000e+01	20.000001	19.296052

```
df[floatcols[20:40]].describe()
```

	v23	v25	v26	v27	v28	v29	v32	v33	v34
count	6.364600e+04	65702.000000	6.448900e+04	6.448900e+04	6.448900e+04	6.448900e+04	6.448900e+04	6.448900e+04	1.142100e+05
mean	1.093088e+00	1.698129	1.876031e+00	2.743454e+00	5.093328e+00	8.206416e+00	1.622151e+00	2.161633e+00	6.406236e+00
std	4.003695e+00	2.956854	5.511957e-01	8.343556e-01	2.677940e+00	1.285432e+00	5.635237e-01	9.848598e-01	2.025179e+00
min	-9.999932e-07	0.041043	-9.346696e-07	-9.915986e-07	-6.960880e-07	-3.040753e-07	-9.559960e-07	-9.713108e-07	-6.707670e-07
25%	-4.093461e-07	0.149666	1.510201e+00	2.196970e+00	3.491553e+00	7.427310e+00	1.262954e+00	1.470096e+00	5.054165e+00
50%	1.845672e-07	0.472307	1.826277e+00	2.673321e+00	5.043831e+00	8.296139e+00	1.557203e+00	1.946169e+00	6.537069e+00
75%	7.809954e-07	1.947268	2.181855e+00	3.224368e+00	6.574727e+00	9.087867e+00	1.896321e+00	2.632911e+00	7.702703e+00
max	2.000000e+01	20.000001	2.000000e+01	2.000000e+01	1.984819e+01	2.000000e+01	1.756098e+01	2.000000e+01	2.000000e+01

v35	v36	v37	v39	v40	v41	v42	v43	v44	v45	v46
6.448900e+04	6.569700e+04	6.447800e+04	6.448500e+04	1.142100e+05	6.448900e+04	6.448900e+04	6.448500e+04	6.452500e+04	6.448900e+04	65702.000000
8.122387e+00	1.337560e+01	7.414708e-01	1.237184e+00	1.046593e+01	7.182551e+00	1.292497e+01	2.216597e+00	1.079517e+01	9.142231e+00	1.630525
1.339802e+00	2.355631e+00	5.413723e-01	2.358155e+00	3.169183e+00	1.004472e+00	9.969760e-01	6.479912e-01	2.110887e+00	2.064508e+00	2.895831
-9.958327e-07	-4.906628e-07	-9.999498e-07	-9.999742e-07	1.238996e-07	-7.272275e-07	-6.206144e-07	-9.724295e-07	-9.482212e-07	-9.202112e-07	0.069349
7.253885e+00	1.176681e+01	4.040413e-01	1.252641e-01	8.408220e+00	6.542650e+00	1.234168e+01	1.787996e+00	9.575104e+00	7.826650e+00	0.123707
8.068506e+00	1.376562e+01	6.428565e-01	3.784183e-01	1.033427e+01	7.203020e+00	1.293436e+01	2.152007e+00	1.078201e+01	9.156046e+00	0.440209
8.936359e+00	1.532153e+01	9.482752e-01	1.192830e+00	1.276518e+01	7.827246e+00	1.349385e+01	2.560679e+00	1.202476e+01	1.041681e+01	1.817872
2.000000e+01	2.000000e+01	2.000000e+01	1.991553e+01	2.000000e+01	2.000000e+01	2.000000e+01	2.000000e+01	1.983168e+01	2.000000e+01	20.000001

`df[floatcols[40:60]].describe()`

	v48	v49	v50	v51	v53	v54	v55	v57	v58	v59
count	6.452500e+04	6.448900e+04	1.142350e+05	6.364300e+04	6.448500e+04	65702.000000	6.448900e+04	6.448900e+04	6.448500e+04	6.452500e+04
mean	1.253802e+01	8.016547e+00	1.504265e+00	7.198159e+00	1.571130e+01	1.253856	1.559556e+00	4.077828e+00	7.701653e+00	1.058794e+01
std	2.196164e+00	9.026805e-01	1.168329e+00	2.510385e+00	7.993680e-01	2.314476	8.343910e-01	6.780415e-01	6.841238e+00	2.071674e+00
min	-9.924422e-07	-6.697975e-07	-9.091393e-07	-3.616122e-07	-9.838107e-07	0.013062	-9.846820e-07	-7.570607e-07	-9.976841e-07	-8.803695e-07
25%	1.121625e+01	7.472959e+00	6.587918e-01	5.597638e+00	1.527540e+01	0.093094	9.817783e-01	3.648892e+00	1.500000e+00	9.055747e+00
50%	1.241161e+01	8.022771e+00	1.211944e+00	7.134018e+00	1.577093e+01	0.313565	1.370940e+00	4.067039e+00	5.330550e+00	1.053511e+01
75%	1.377798e+01	8.558951e+00	2.007189e+00	8.643772e+00	1.621687e+01	1.406020	1.944445e+00	4.488055e+00	1.396450e+01	1.203309e+01
max	2.000000e+01	2.000000e+01	2.000000e+01	2.000000e+01	2.000000e+01	20.000001	2.000000e+01	2.000000e+01	2.000000e+01	2.000000e+01

v60	v61	v63	v64	v65	v67	v68	v69	v70	v73
6.448900e+04	6.452500e+04	65702.000000	6.452500e+04	64481.000000	6.448900e+04	64485.000000	6.442600e+04	65685.000000	6.448500e+04
1.714294e+00	1.458303e+01	1.687327	6.343713e+00	15.847557	9.287275e+00	17.564117	9.449335e+00	12.269960	2.433303e+00
5.376079e-01	2.120976e+00	2.967313	2.525594e+00	1.878107	1.123350e+00	2.289924	1.900495e+00	2.314462	7.986390e-01
-9.970164e-07	-6.113428e-07	0.053055	-8.770837e-07	0.659296	-2.413161e-07	1.501359	-9.920262e-07	0.427095	-9.838592e-07
1.361526e+00	1.360481e+01	0.142445	4.785218e+00	15.033712	8.577383e+00	17.077177	8.389830e+00	10.802010	1.902173e+00
1.667033e+00	1.507589e+01	0.460832	6.108900e+00	16.262525	9.313861e+00	18.274548	9.516129e+00	12.494872	2.331261e+00
2.010534e+00	1.610748e+01	1.848941	7.518160e+00	17.164253	9.992977e+00	18.911391	1.053763e+01	13.994624	2.850558e+00
2.000000e+01	1.884696e+01	20.000001	2.000000e+01	20.000001	2.000000e+01	20.000001	2.000000e+01	19.816311	2.000000e+01

`df[floatcols[60:80]].describe()`

	v76	v77	v78	v80	v81	v82	v83	v84	v85	v86
count	6.452500e+04	6.448900e+04	6.442600e+04	6.447000e+04	6.569700e+04	6.569700e+04	6.448900e+04	6.448900e+04	6.363900e+04	6.448900e+04
mean	2.405056e+00	7.307366e+00	1.333448e+01	2.209700e+00	7.287174e+00	6.208356e+00	2.173808e+00	1.607956e+00	2.822253e+00	1.220184e+00
std	1.383719e+00	1.256062e+00	1.843919e+00	1.074983e+00	2.223637e+00	3.678042e+00	1.062289e+00	9.412067e-01	1.423208e+00	4.658070e-01
min	-6.171967e-07	-7.729896e-07	-9.902572e-07	-9.992919e-07	-1.443765e-07	-7.251767e-07	-9.865679e-07	-9.994245e-07	-9.995741e-07	-9.814310e-07
25%	1.627384e+00	6.504957e+00	1.231343e+01	1.446541e+00	5.983212e+00	3.385300e+00	1.432761e+00	9.565997e-01	1.836735e+00	9.370128e-01
50%	2.170300e+00	7.375825e+00	1.333333e+01	2.090909e+00	7.517742e+00	3.688459e+00	1.937244e+00	1.423295e+00	2.594937e+00	1.158301e+00
75%	2.806533e+00	8.163901e+00	1.439394e+01	2.857142e+00	8.776052e+00	8.791594e+00	2.668161e+00	2.069454e+00	3.574468e+00	1.422051e+00
max	2.000000e+01	1.597351e+01	2.000000e+01	2.000000e+01	2.000000e+01	2.000000e+01	2.000000e+01	2.000000e+01	2.000000e+01	1.756098e+01

v87	v88	v89	v90	v92	v93	v94	v95	v96	v97
65658.000000	6.448900e+04	65702.000000	6.448500e+04	6.447800e+04	6.448900e+04	6.448900e+04	6.447800e+04	6.448900e+04	6.447800e+04
10.180216	1.924184e+00	1.518425	9.669126e-01	5.823668e-01	5.475185e+00	3.852883e+00	6.657576e-01	6.457952e+00	7.622554e+00
3.000053	1.048548e+00	2.812902	1.789302e-01	2.402102e-01	1.640349e+00	8.549950e-01	2.641127e-01	1.120472e+00	1.924074e+00
0.872396	-9.990082e-07	0.022365	-2.343004e-07	3.393210e-08	4.251919e-07	-9.687809e-07	-5.784674e-07	-8.832504e-07	-9.059520e-07
8.055425	1.180276e+00	0.099645	8.627289e-01	4.350468e-01	4.548473e+00	3.333932e+00	5.016117e-01	5.758412e+00	6.315790e+00
9.995718	1.761547e+00	0.332364	9.691829e-01	5.426694e-01	5.301047e+00	3.743106e+00	6.201132e-01	6.511627e+00	7.446808e+00
12.230477	2.461539e+00	1.746064	1.061779e+00	6.783547e-01	6.223883e+00	4.225063e+00	7.691490e-01	7.230769e+00	8.780488e+00
19.842754	2.000000e+01	20.000001	6.305775e+00	8.923843e+00	2.000000e+01	1.901631e+01	9.070538e+00	2.000000e+01	2.000000e+01

```
df[floatcols[80:100]].describe()
```

	v98	v99	v100	v101	v102	v103	v104	v105	v106	v108
count	6.566700e+04	6.448900e+04	6.448500e+04	6.452500e+04	6.300500e+04	6.448900e+04	6.448900e+04	65663.000000	6.452500e+04	6.569700e+04
mean	7.667624e+00	1.250721e+00	1.209162e+01	6.866414e+00	2.890289e+00	5.296716e+00	2.642828e+00	1.081045	1.179136e+01	2.152620e+00
std	2.325863e+00	4.614097e-01	6.888297e+00	2.354673e+00	1.824041e+00	1.228804e+00	8.857686e-01	2.247308	2.954102e+00	9.131328e-01
min	-5.605928e-07	-9.787309e-07	-9.981311e-07	-3.942767e-07	-7.111493e-07	-9.757743e-07	-8.683169e-07	0.000094	-5.467029e-07	4.251643e-07
25%	6.125288e+00	9.330819e-01	5.754166e+00	5.257973e+00	1.793632e+00	4.500904e+00	2.140932e+00	0.055565	1.004981e+01	1.530367e+00
50%	7.644154e+00	1.235546e+00	1.447594e+01	6.623714e+00	2.462898e+00	5.125847e+00	2.512034e+00	0.242556	1.208518e+01	1.980834e+00
75%	9.064070e+00	1.552500e+00	1.831746e+01	8.240794e+00	3.413256e+00	5.874545e+00	2.948754e+00	1.020948	1.376751e+01	2.541602e+00
max	1.905880e+01	2.000000e+01	2.000000e+01	2.000000e+01	2.000000e+01	1.877525e+01	2.000000e+01	20.000001	2.000000e+01	2.000000e+01

v109	v111	v114	v115	v116	v117	v118	v119	v120	v121
6.569700e+04	6.448900e+04	1.142910e+05	6.442600e+04	6.448500e+04	6.569700e+04	6.447800e+04	6.364100e+04	6.448500e+04	6.448100e+04
4.181284e+00	3.365314e+00	1.357445e+01	1.054805e+01	2.291218e+00	8.303857e+00	8.364651e+00	3.168970e+00	1.291218e+00	2.737596e+00
3.712001e+00	1.487422e+00	2.613221e+00	1.901484e+00	6.702714e-01	3.618001e+00	2.002097e+00	4.240120e+00	7.383739e-01	1.356294e+00
-9.996884e-07	-9.990270e-07	-9.335327e-10	-9.853189e-07	-9.450359e-07	-9.991992e-07	-1.695463e-07	-9.998183e-07	-9.932534e-07	-9.820642e-07
1.842106e+00	2.317597e+00	1.199531e+01	9.457365e+00	1.851474e+00	5.763345e+00	6.981627e+00	3.150087e-01	7.758620e-01	1.786965e+00
3.087909e+00	3.108809e+00	1.403998e+01	1.047619e+01	2.222223e+00	8.070175e+00	8.136964e+00	1.461561e+00	1.144708e+00	2.436195e+00
5.151941e+00	4.120879e+00	1.537267e+01	1.161290e+01	2.649681e+00	1.050322e+01	9.565218e+00	4.166049e+00	1.647220e+00	3.379175e+00
2.000000e+01	2.000000e+01	2.000000e+01	2.000000e+01	2.000000e+01	2.000000e+01	2.000000e+01	2.000000e+01	1.039427e+01	2.000000e+01

Out[17]:

```
df[floatcols[100:109]].describe()
```

	v122	v123	v124	v126	v127	v128	v130	v131
count	6.447000e+04	63643.000000	6.570200e+04	6.448900e+04	6.448900e+04	6.569700e+04	6.447800e+04	6.442600e+04
mean	6.822439e+00	3.549938	9.198120e-01	1.672658e+00	3.239542e+00	2.030373e+00	1.925763e+00	1.739389e+00
std	1.795978e+00	2.604704	2.099407e+00	5.031683e-01	1.625988e+00	1.074232e+00	1.264497e+00	1.134702e+00
min	-9.978497e-07	0.019139	-9.994953e-07	-9.564174e-07	-9.223798e-07	8.197812e-07	-9.901257e-07	-9.999134e-07
25%	5.647712e+00	1.963315	2.053777e-02	1.417600e+00	2.101900e+00	1.393830e+00	1.106172e+00	1.012658e+00
50%	6.749117e+00	2.739239	1.398639e-01	1.614802e+00	2.963620e+00	1.798436e+00	1.560138e+00	1.589403e+00
75%	7.911392e+00	4.075361	8.718333e-01	1.843886e+00	4.108146e+00	2.390158e+00	2.332425e+00	2.261905e+00
max	2.000000e+01	19.686069	2.000000e+01	1.563161e+01	2.000000e+01	2.000000e+01	2.000000e+01	2.000000e+01

NULL/missing value percentage of each column:

```
nullcol = list()
print("Number of columns with null value:",end=" ")
for col in df:
    if(df[col].isna().sum())!=0:
        nullcol.append(col)
print(len(nullcol),end="\n\n")
print("Number of columns with no null value:",end=" ")
nonnullcol = list()
for col in df:
    if(df[col].isna().sum())==0:
        nonnullcol.append(col)
print(len(nonnullcol),end="\n\n")
print("Columns with null value(percentage):")
for col in df:
    if(df[col].isna().sum())!=0:
        print(col,round(((df[col].isna().sum())/df.shape[0])*100,2),end=" // ")
```

Number of columns with null value: 119

Number of columns with no null value: 14

Columns with null value(percentage):

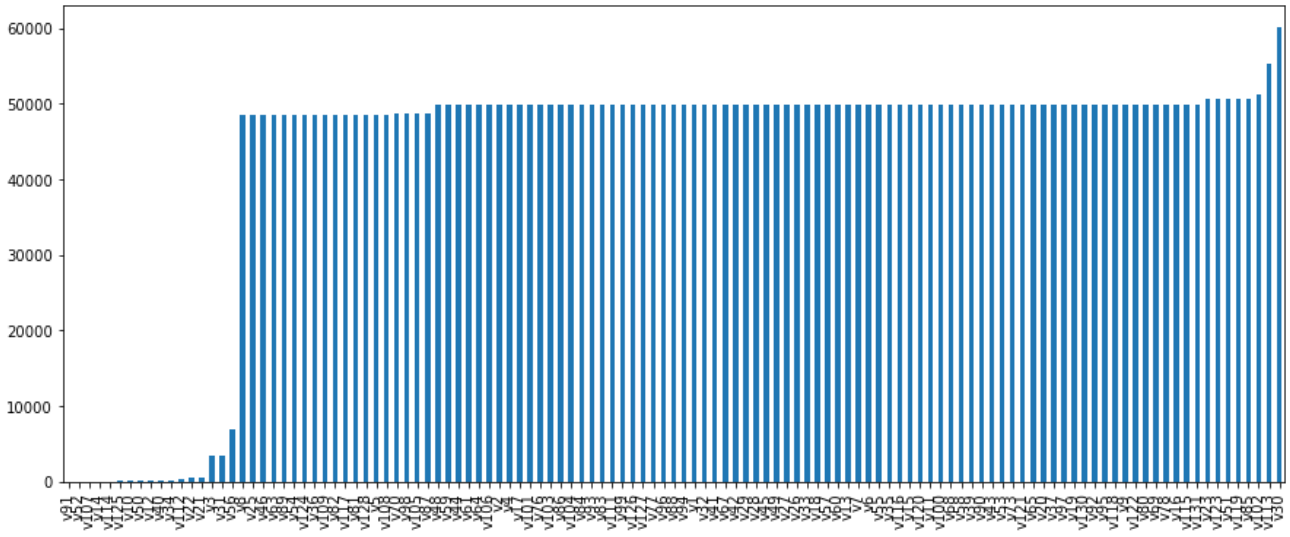
v1 43.59 // v2 43.56 // v3 3.02 // v4 43.56 // v5 42.53 // v6 43.59 // v7 43.59 // v8 42.53 // v9 43.61 // v10 0.07 // v11 43.59 // v12 0.08 // v13 43.59 // v14 0.0 // v15 43.59 // v16 43.64 // v17 43.56 // v18 43.59 // v19 43.6 // v20 43.6 // v21 0.53 // v22 0.44 // v23 44.33 // v25 42.53 // v26 43.59 // v27 43.59 // v28 43.59 // v29 43.59 // v30 52.58 // v31 3.02 // v32 43.59 // v33 43.59 // v34 0.1 // v35 43.59 // v36 42.53 // v37 43.6 // v39 43.59 // v40 0.1 // v41 43.59 // v42 43.59 // v43 43.59 // v44 43.56 // v45 43.59 // v46 42.53 // v48 43.56 // v49 43.59 // v50 0.08 // v51 44.33 // v52 0.0 // v53 43.59 // v54 42.53 // v55 43.59 // v56 6.02 // v57 43.59 // v58 43.59 // v59 43.56 // v60 43.59 // v61 43.56 // v63 42.53 // v64 43.56 // v65 43.6 // v67 43.59 // v68 43.59 // v69 43.64 // v70 42.54 // v73 43.59 // v76 43.56 // v77 43.59 // v78 43.64 // v80 43.61 // v81 42.53 // v82 42.53 // v83 43.59 // v84 43.59 // v85 44.33 // v86 43.59 // v87 42.57 // v88 43.59 // v89 42.53 // v90 43.59 // v91 0.0 // v92 43.6 // v93 43.59 // v94 43.59 // v95 43.6 // v96 43.59 // v97 43.6 // v98 42.56 // v99 43.59 // v100 43.59 // v101 43.56 // v102 44.89 // v103 43.59 // v104 43.59 // v105 42.56 // v106 43.56 // v107 0.0 // v108 42.53 // v109 42.53 // v111 43.59 // v112 0.33 // v113 48.38 // v114 0.03 // v115 43.64 // v116 43.59 // v117 42.53 // v118 43.6 // v119 44.33 // v120 43.59 // v121 43.6 // v122 43.61 // v123 44.33 // v124 42.53 // v125 0.07 // v126 43.59 // v127 43.59 // v128 42.53 // v130 43.6 // v131 43.64 //

So, there a large number of columns having missing value percentage greater than 40.

Null value count graph:

plot of null value counts of columns to see the distribution of null values

```
print("Null value count graph")
missing = df.isnull().sum()
missing = missing[missing > 0]
missing.sort_values(inplace=True)
missing.plot.bar(figsize=(15,6))
plt.show()
```



So, there are 102 columns having missing value percentage greater than 40.

Unique Value Count:

For Integer Columns:

```
print("Number of integer columns:",intcols.shape[0])
print("Unique value count of each integer column:")
for col in df[intcols]:
   print(col," : ",df[col].nunique())
```

Number of integer columns: 6

Unique value count of each integer column:

ID : 114321

target : 2

v38 : 12

v62 : 8

v72 : 13

v129 : 10

For Float columns:

```
print("Number of floating columns:",floatcols.shape[0])
print("Unique value count of each float64 column:")
for col in df[floatcols]:
   print(col," : ",df[col].nunique(),end=' ')
```

Number of floating columns: 108

Unique value count of each float64 column:

v1 : 64487 v2 : 64524 v4 : 64524 v5 : 65671 v6 : 64487 v7 : 64489

v8 : 65688 v9 : 64451 v10 : 112485 v11 : 64477 v12 : 114233

v13 : 64488 v14 : 110472 v15 : 64481 v16 : 64414 v17 : 64525

v18 : 64489 v19 : 64478 v20 : 64469 v21 : 113709 v23 : 63638

v25 : 65679 v26 : 64489 v27 : 64489 v28 : 64482 v29 : 64487

v32 : 64488 v33 : 64489 v34 : 114209 v35 : 64489 v36 : 65609

v37 : 64474 v39 : 64484 v40 : 114192 v41 : 64488 v42 : 64472

v43 : 64484 v44 : 64520 v45 : 64486 v46 : 65697 v48 : 64519

v49 : 64487 v50 : 114224 v51 : 63637 v53 : 64469 v54 : 65694

v55 : 64488 v57 : 64488 v58 : 64482 v59 : 64524 v60 : 64488

v61 : 64519 v63 : 65681 v64 : 64524 v65 : 64472 v67 : 64487

v68 : 64474 v69 : 64332 v70 : 65631 v73 : 64484 v76 : 64525

v77 : 64489 v78 : 64265 v80 : 64466 v81 : 65685 v82 : 65663

v83 : 64489 v84 : 64487 v85 : 63637 v86 : 64489 v87 : 65630

v88 : 64489 v89 : 65686 v90 : 64484 v92 : 64478 v93 : 64489

v94 : 64489 v95 : 64478 v96 : 64489 v97 : 64470 v98 : 65656

v99 : 64488 v100 : 64471 v101 : 64525 v102 : 62986 v103 : 64489

v104 : 64489 v105 : 65658 v106 : 64521 v108 : 65687 v109 : 65673

v111 : 64486 v114 : 114267 v115 : 64253 v116 : 64485 v117 : 65662

v118 : 64470 v119 : 63638 v120 : 64484 v121 : 64480 v122 : 64461

v123 : 63641 v124 : 65691 v126 : 64487 v127 : 64488 v128 : 65687

v130 : 64476 v131 : 64414

Name, type(continuous/categorical),how much percentage is null, how many outliers

5 number statistics

[(Source of data)

(Name,type,null percentage, how many outliers)

(5number statistics : min, max, q1,q2,q3)

]

Exploratory Data Analysis

Data Visualization tools are of great importance in the analytics industry as they give a clear idea of the complex data involved. Python is one of the most popular languages for visualization with its variety of tools.

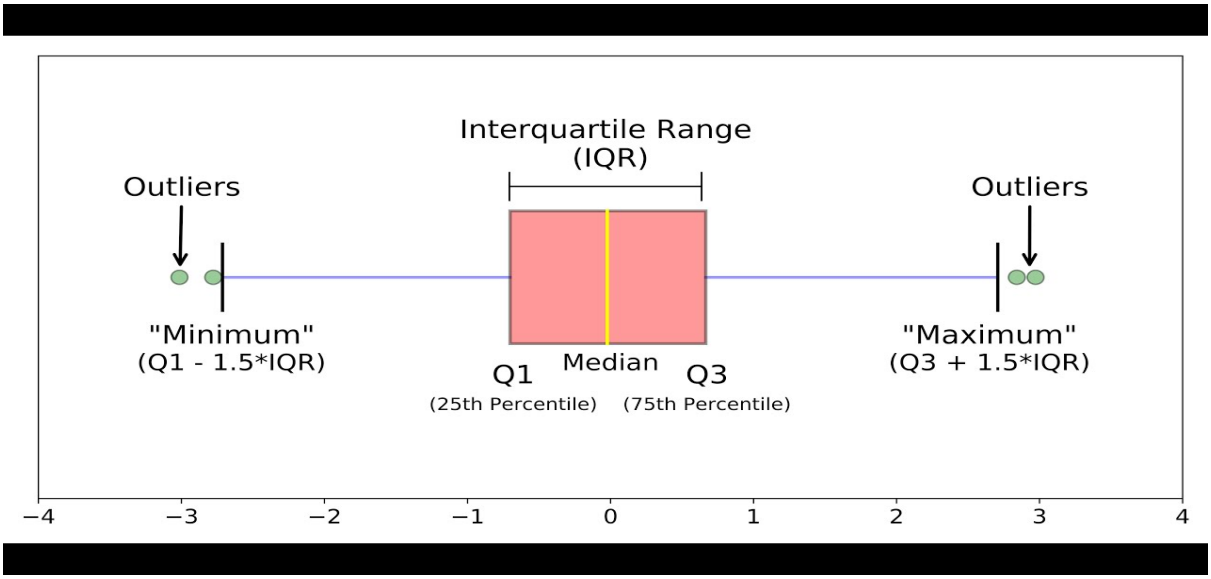
Two of Python's greatest visualization tools are **Matplotlib** and **Seaborn**. Seaborn library is basically based on Matplotlib.

Matplotlib: Matplotlib is mainly deployed for basic plotting. Visualization using Matplotlib generally consists of bars, pies, lines, scatter plots and so on.

Seaborn: Seaborn, on the other hand, provides a variety of visualization patterns. It uses fewer syntax and has easily interesting default themes. It specializes in statistics visualization and is used if one has to summarize data in visualizations and also show the distribution in the data.

BOXPLOT:

Box plot is method to graphically show the spread of a numerical variable through quartiles. A function seaborn.**boxplot**() is used to plot the boxplot. From the below image you can see what information we generally get from a box plot.

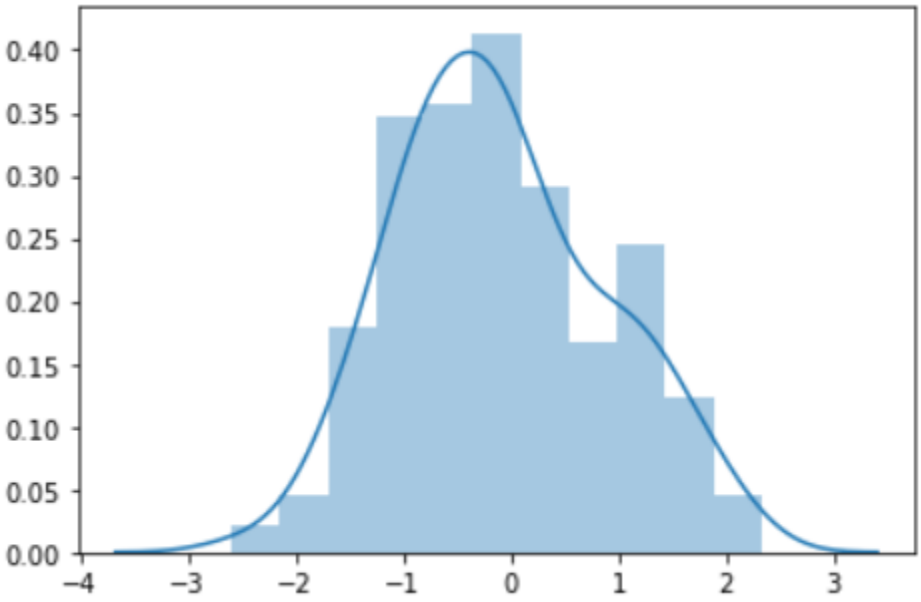


The Top of the (red) box is the 25% percentile and the bottom is the 75% percentile value of the data. So, essentially the box represents the **middle 50% of all the datapoints** which represents the core region when the data is situated. The height of the boxplot is also called the **Inter Quartile Range (IQR)**, which mathematically is the difference between the 75th and 25th percentile values of the data.

The thick yellow line in the middle of the box represents the median. Whereas, the upper and lower maximum and minimum marks 1.5 times the IQR from the top (and bottom) of the box. **The points that lie outside the whiskers, that is, (1.5 x IQR) in both directions are generally considered as outliers**.

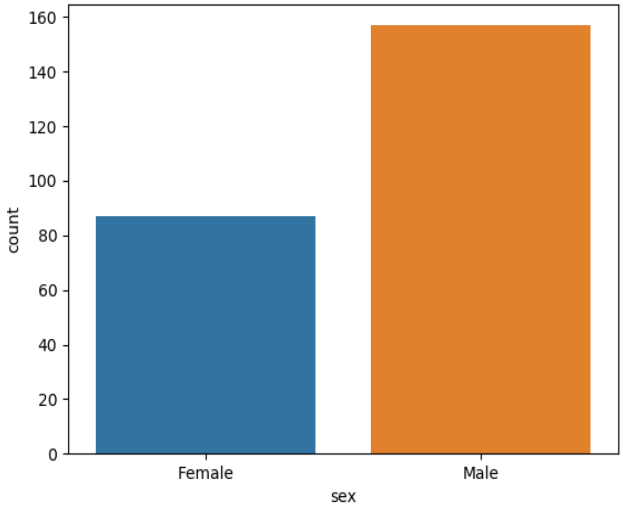
DISTPLOT:

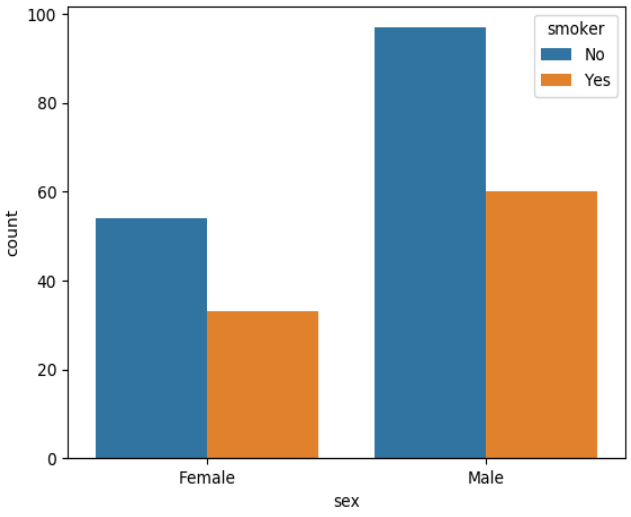
Python Seaborn module contains various functions to plot the data and depict the data variations. **seaborn.distplot**() function is used to plot the **distplot**. The **distplot** represents the univariate distribution of data i.e. data distribution of a variable against the density distribution.



COUNTPLOT:

seaborn.**countplot**() function is used to plot countplot. A count plot can be thought of as a histogram across a categorical, instead of quantitative, variable.

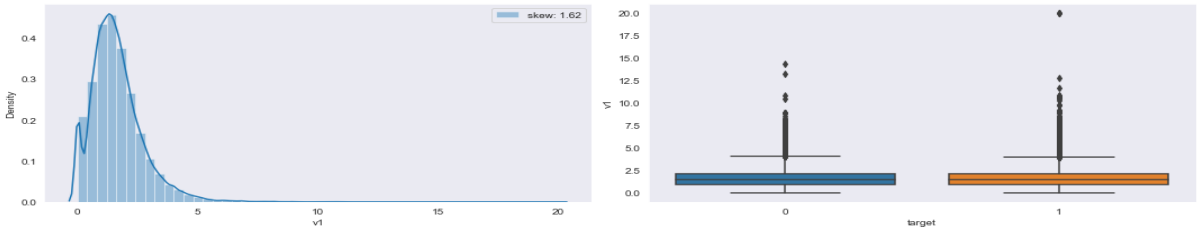




Floating point Columns:

Col_name: v1

Pyplot representation of v1:



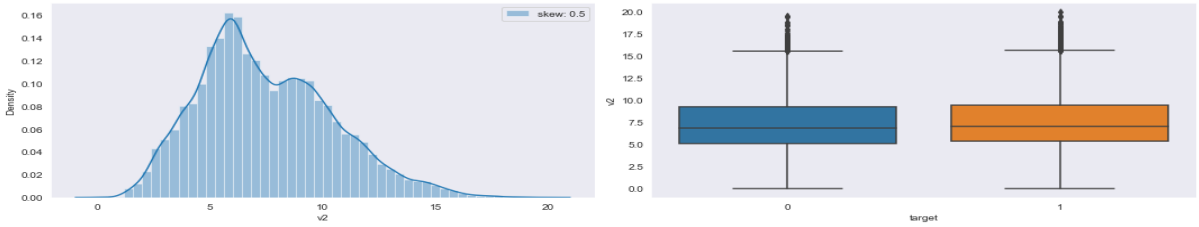
5-number statistics values of v1:

v1 Min: -9.99649701427e-07 Q1: 0.913579829645 Q2: 1.46954989655 Q3: 2.1361279139500002 Max: 20.000000629400002

Conclusion: From the density curve we observed that the distribution of column v1 is negligibly positively/right skewed. Whereas from boxplot it reflects that there is a large number of outliers ([q3+1.5*IQR] region only) present in v1.

Col_name: v2

Pyplot representation of v2:



5-number statistics values of v2:

v2 Min: -9.81761449207e-07 Q1: 5.316428241050001 Q2: 7.023803118049999 Q3: 9.46549707005 Max: 19.999999908699998

Conclusion: The distribution of column v2 is negligibly positive/right skewed. Whereas the boxplot reflects a large number of outliers([q3+1.5*IQR] region only)present in column v2.

Col_name: v4

Pyplot representation of v4:

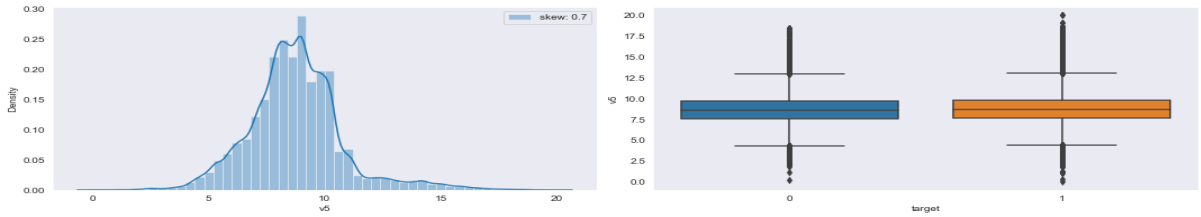
5-number statistics values of v4:

v4 Min: -6.47592938974e-07 Q1: 3.4873981780900003 Q2: 4.20599078916 Q3: 4.83325029006 Max: 19.9999997446

Conclusion: The distribution of column v4 is negligibly negative/left skewed. Whereas the boxplot reflects a large number of outliers(both [q1-1.5*IQR] & [q3+1.5*IQR]) present in v4.

Col_name: v5

Pyplot representation of v5:



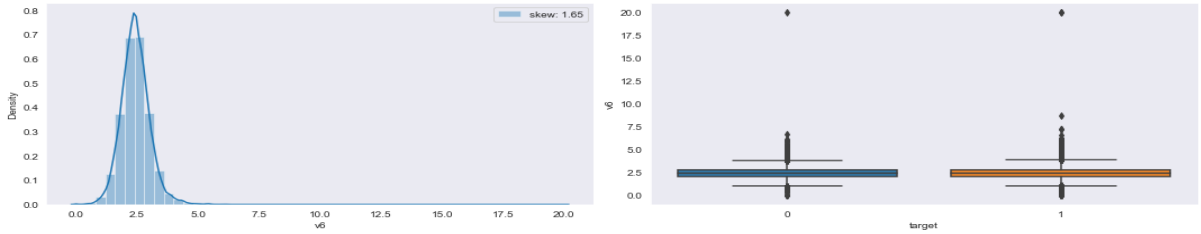
5-number statistics values of v5:

v5 Min: -5.28706773973e-07 Q1: 7.6059176934899995 Q2: 8.67086715203 Q3: 9.77135265768 Max: 20.000000353900003

Conclusion: The distribution of column v5 is negligibly positive/right skewed. Whereas in boxplot a large number of outliers(both [q1-1.5*IQR] & [q3+1.5*IQR]) present in v5.

Col_name: v6

Pyplot representation of v6:



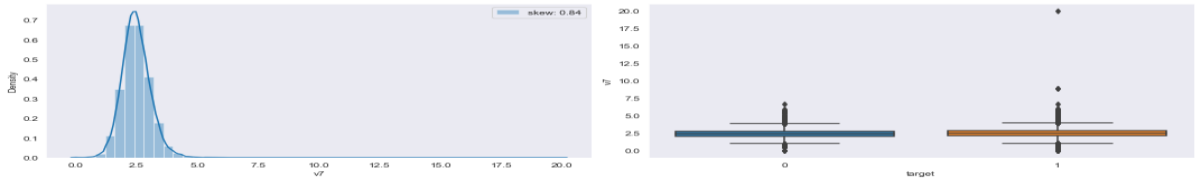
5-number statistics values of v6:

v6 Min: -9.055091449979999e-07 Q1: 2.06506390521 Q2: 2.41279049981 Q3: 2.77528469464 Max: 20.0000005964

Conclusion: The distribution of column v6 is slightly positive/right skewed. Whereas in boxplot a large number of outliers (both [q1-1.5*IQR] & [q3+1.5*IQR]) present in v6.

Col_name: v7

Pyplot representation of v7:



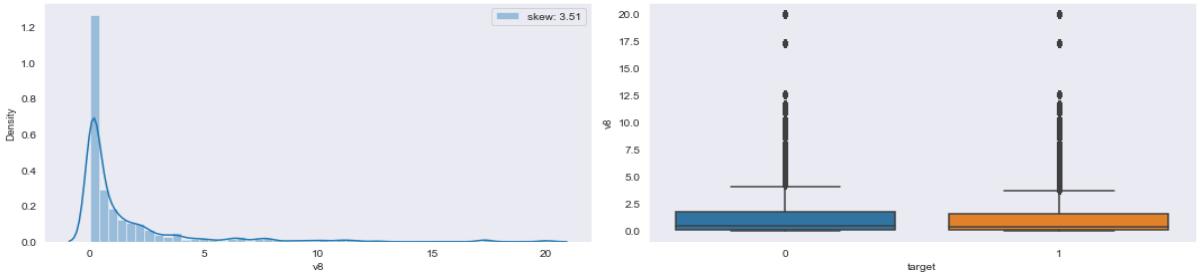
5-number statistics values of v7:

v7 Min: -9.468765496739999e-07 Q1: 2.10147749588 Q2: 2.4521664187 Q3: 2.83428521518 Max: 19.9999998141

Conclusion: The distribution of column v7 is slightly positive/right skewed. Whereas in boxplot a large number of outliers(both [q1-1.5*IQR] & [q3+1.5*IQR]) present in v7

Col_name: v8

Pyplot representation of v8:



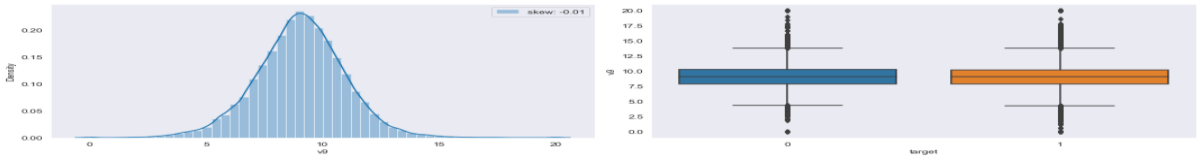
5-number statistics values of v8:

v8 Min: -7.783778107579999e-07 Q1: 0.08658986231387501 Q2: 0.38603171417099996 Q3: 1.6252458337375 Max: 20.000000997

Conclusion: The distribution of column v8 is quite high positive/right skewed. Whereas from boxplot it reflects that there is a large number of outliers (q3+1.5*IQR region only) present in v8 mainly high at maximum point w.r.t '0' and '1' case.

Col_name: v9

Pyplot representation of v9:



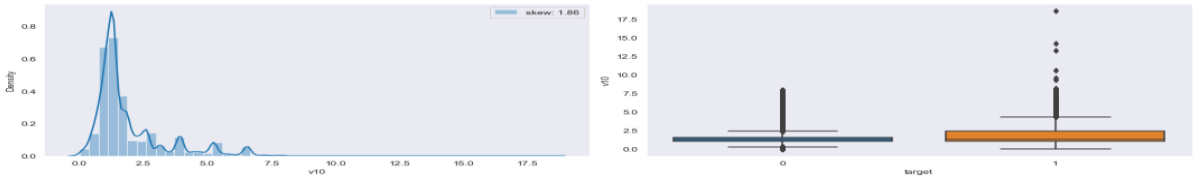
5-number statistics values of v9:

v9 Min: -9.82875684434e-07 Q1: 7.8536589573599995 Q2: 9.05958234041 Q3: 10.232558533275 Max: 20.000000750199998

Conclusion: The distribution of column v9 is slightly negative/left skewed. Whereas from boxplot it reflects that there is a large number of outliers(both [q1-1.5*IQR]&[q3+1.5*IQR]) present in v9 with equal % at MAX and MIN points of '0' and '1' case.

Col_name: v10

Pyplot representation of v10:



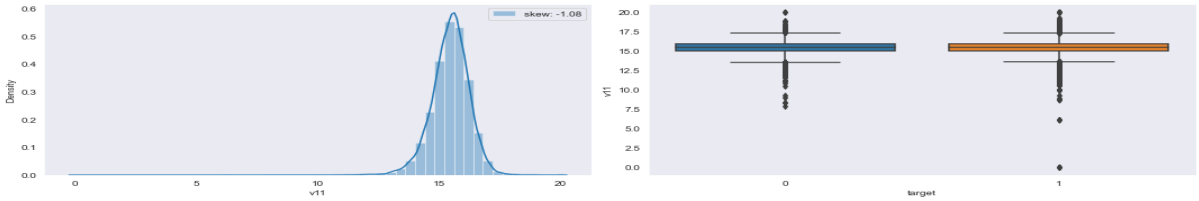
5-number statistics values of v10:

v10 Min: -9.87531659989e-07 Q1: 1.05032820389 Q2: 1.31291009339 Q3: 2.1006572207599996 Max: 18.5339164478

Conclusion: The distribution of column v10 is highly positive/right skewed. Whereas in boxplot a large number of outliers(both [q1-1.5*IQR] & [q3+1.5*IQR]) present in v5respectively high in first topmost region compared to second bottom region.

Col name: v11

Pyplot representation of v11:



5-number statistics values of v11:

v11 Min: -1.45906182755e-07 Q1: 15.0000005446 Q2: 15.495951684000001 Q3: 15.949999656700001 Max: 20.0000009233

Conclusion: The distribution of column v11 is slightly negative/left skewed. Whereas in boxplot a large number of outliers(both [q1-1.5*IQR] & [q3+1.5*IQR])present in v11 respectively high in second bottom region compared to first topmost region.

Col name: v12

Pyplot representation of v12:

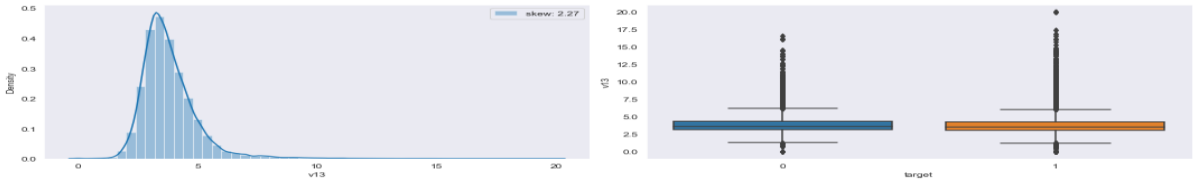
5-number statistics values of v12:

v12 Min: 5.14322389107e-07 Q1: 6.322470981605 Q2: 6.61296881023 Q3: 7.019982793805 Max: 18.710550390599998

Conclusion: The distribution of column v12 is highly positive/right skewed. Whereas in boxplot a large number of outliers(both [q1-1.5*IQR] & [q3+1.5*IQR])present in v12 mainly in '1' case means delay in processing claims.

Col name: v13

Pyplot representation of v13:



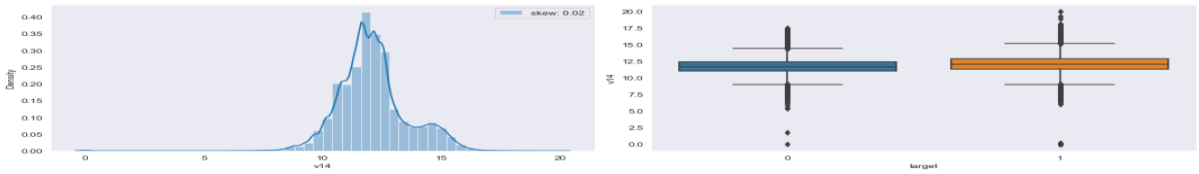
5-number statistics values of v13:

v13 Min: -8.46488878755e-07 Q1: 3.06799817733 Q2: 3.5910809381199997 Q3: 4.28948559252 Max: 20.0000009059

Conclusion: The distribution of column v13 is highly positive/right skewed. Whereas in boxplot a large number of outliers(both [q1-1.5*IQR] & [q3+1.5*IQR])present in v12 mainly high in '1' case.

Col_name: v14

Pyplot representation of v14:



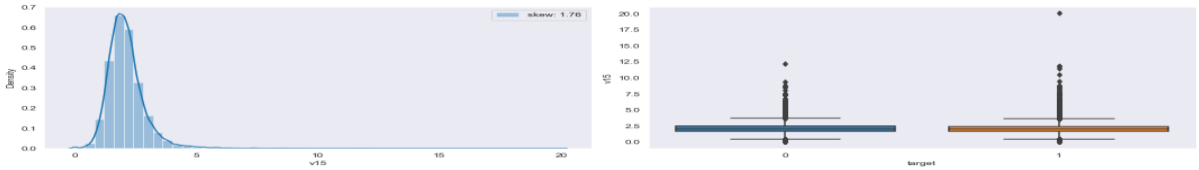
5-number statistics values of v14:

v14 Min: -9.73883146957e-07 Q1: 11.2560172864 Q2: 11.9678254625 Q3: 12.7157742607 Max: 19.9999996125

Conclusion: The distribution of column v14 is quite high positive/right skewed. Whereas in boxplot a large number of outliers (both [q1-1.5*IQR] & [q3+1.5*IQR]) present in v14 mainly high in both '0' and '1' case.

Col_name: v15

Pyplot representation of v15:



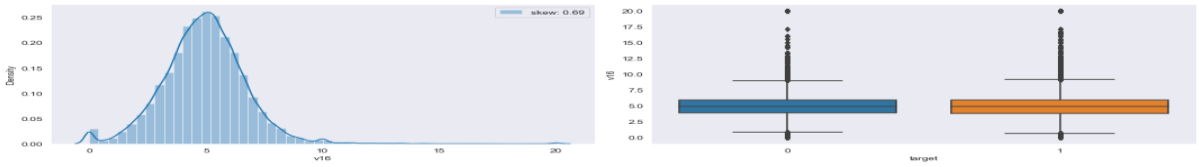
5-number statistics values of v15:

v15 Min: -8.830426652450001e-07 Q1: 1.61141878911 Q2: 1.9920311905599999 Q3: 2.4186763298900003 Max: 19.9999990205

Conclusion: The distribution of column v15 is highly positive/right skewed. Whereas in boxplot a large number of outliers (both [q1-1.5*IQR] & [q3+1.5*IQR]) present in v15 with higher % at MAX points w.r.t '0' and '1' case.

Col_name: v16

Pyplot representation of v16:



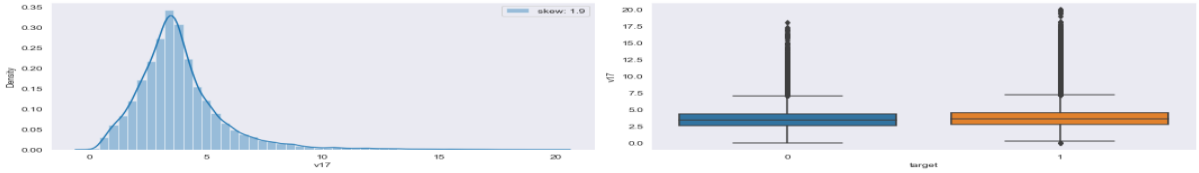
5-number statistics values of v16:

v16 Min: -9.978294167489999e-07 Q1: 3.8647345326224998 Q2: 4.932126795555 Q3: 5.957447683269999 Max: 20.0000009395

Conclusion: The distribution of column v16 is slightly negative/left skewed. Whereas in boxplot a large number of outliers (both [q1-1.5*IQR] & [q3+1.5*IQR]) present in v16 with higher % at MAX points w.r.t '0' and '1' case.

Col_name: v17

Pyplot representation of v17:



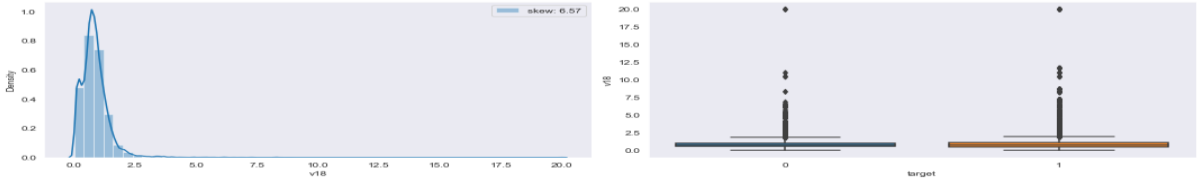
5-number statistics values of v17:

v17 Min: -9.066455398280001e-07 Q1: 2.6991224419 Q2: 3.55426683121 Q3: 4.512220597200001 Max: 19.9999992246

Conclusion: The distribution of column v17 is slightly negative/left skewed. Whereas in boxplot a large number of outliers ([q3+1.5*IQR] region only) present in v17 with higher % at MAX points w.r.t '0' and '1' case.

Col_name: v18

Pyplot representation of v18:



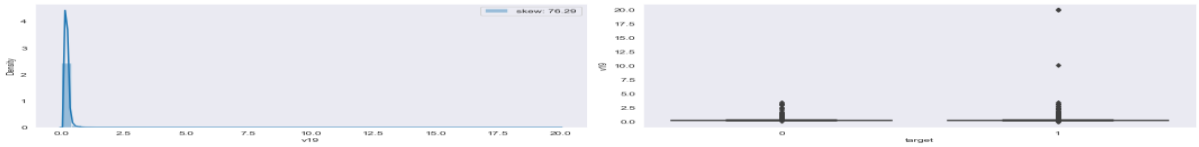
5-number statistics values of v18:

v18 Min: 4.4754697916000003e-07 Q1: 0.509576725815 Q2: 0.7739062613269999 Q3: 1.07145323702 Max: 20.0000009159

Conclusion: The distribution of column v18 is slightly positive/right skewed. Whereas in boxplot a large number of outliers ([q3+1.5*IQR] region only) present in v18 with higher % at MAX points w.r.t '0' and '1' case.

Col_name: v19

Pyplot representation of v19:



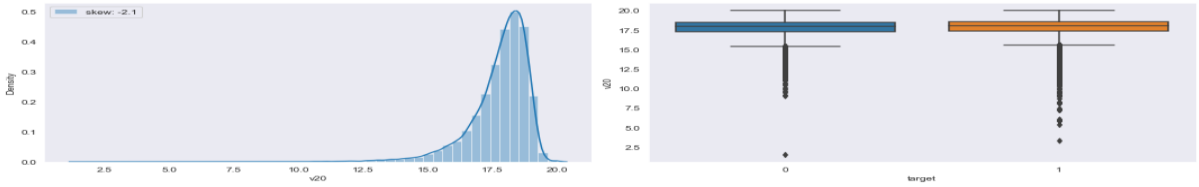
5-number statistics values of v19:

v19 Min: -5.178986590640001e-07 Q1: 0.17391538694125 Q2: 0.1986553651505 Q3: 0.238631616223 Max: 20.0000007723

Conclusion: The distribution of column v19 is highly positive/right skewed. Whereas in boxplot a very large number of outliers ([q3+1.5*IQR] region only) present in v19 with very much higher % at MAX points w.r.t '0' and '1' case.

Col_name:v20

Pyplot representation of v20:



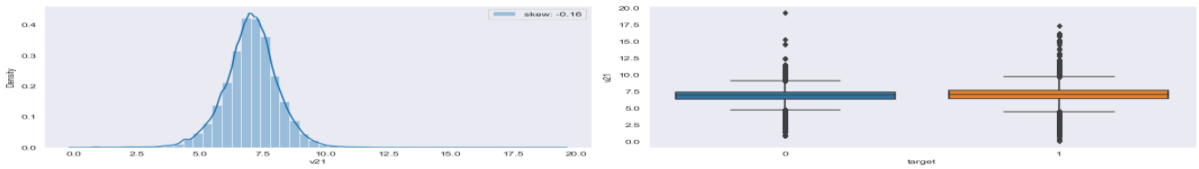
5-number statistics values of v20:

v20 Min: 1.51677642278 Q1: 17.3307026766 Q2: 18.0365851967 Q3: 18.5426313097 Max: 20.0000009905

Conclusion: The distribution of column v20 is highly negative/left skewed. Whereas in boxplot a very large number of outliers ([q1-1.5*IQR] region only) present in v20 with very much higher % at MIN points w.r.t '0 ' and '1' case.

Col_name:v21

Pyplot representation of v21:



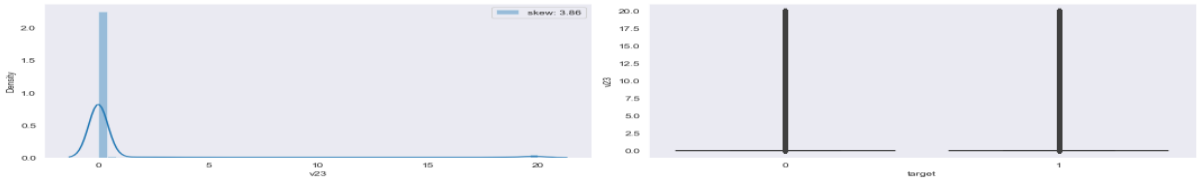
5-number statistics values of v21:

v21 Min: 0.106180586922 Q1: 6.415534721427501 Q2: 7.04541636615 Q3: 7.670577316285 Max: 19.2960519612

Conclusion: The distribution of column v21 is slightly negative/left skewed. Whereas in boxplot a very large number of outliers (both[q1-1.5*IQR] & [q3+1.5*IQR]) present in v21 mainly high at both MAX and MIN poi nts in both '0' and '1' case.

Col_name:v23

Pyplot representation of v23:



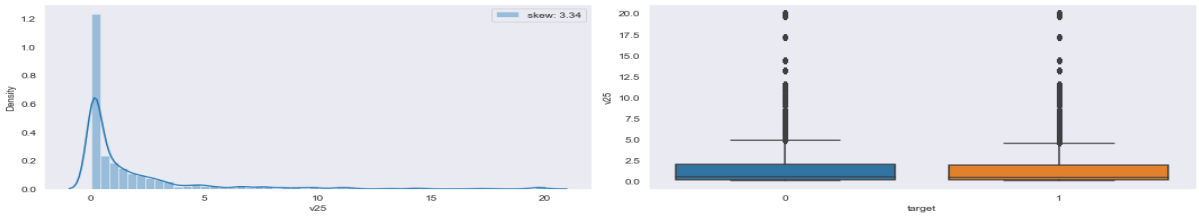
5-number statistics values of v23:

v23 Min: -9.99993198897e-07 Q1: -4.09346082682e-07 Q2: 1.8456722467149999e-07 Q3: 7.80995437008e-07 Max: 20.0 000009982

Conclusion: The distribution of column v23 is slightly positive/right skewed. Whereas in boxplot a very huge number of outliers (both[q1-1.5*IQR] & [q3+1.5*IQR]) present in v23 with very much higher % at MAX poin ts w.r.t '0' and '1' case.

Col_name:v25

Pyplot representation of v25:



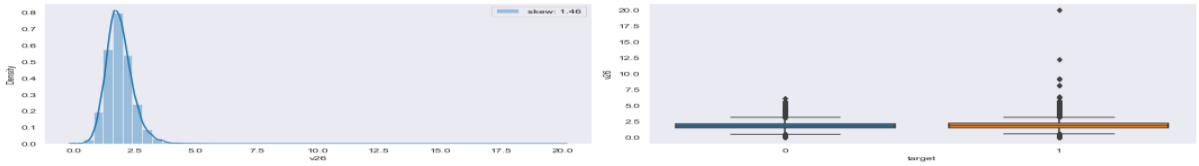
5-number statistics values of v25:

v25 Min: 0.0410432424479 Q1: 0.14966632394775 Q2: 0.4723070845245 Q3: 1.9472677508924998 Max: 20.0000009994

Conclusion: The distribution of column v25 is highly positive/right skewed. Whereas in boxplot a very huge number of outliers (both[q1-1.5*IQR] & [q3+1.5*IQR]) present in v25 with higher % at MAX points w.r.t '0' and '1' case.

Col_name:v26

Pyplot representation of v26:



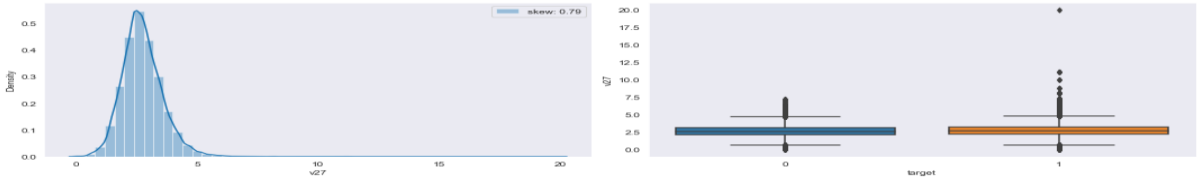
5-number statistics values of v26:

v26 Min: -9.34669608422e-07 Q1: 1.5102007869499998 Q2: 1.8262765996099999 Q3: 2.18185548638 Max: 19.9999996744

Conclusion: The distribution of column v26 is quite high positive/right skewed. Whereas in boxplot a very huge number of outliers (both[q1-1.5*IQR] & [q3+1.5*IQR]) present in v26 with higher % at MAX points w.r.t '0' and '1' case.

Col_name:v27

Pyplot representation of v27:



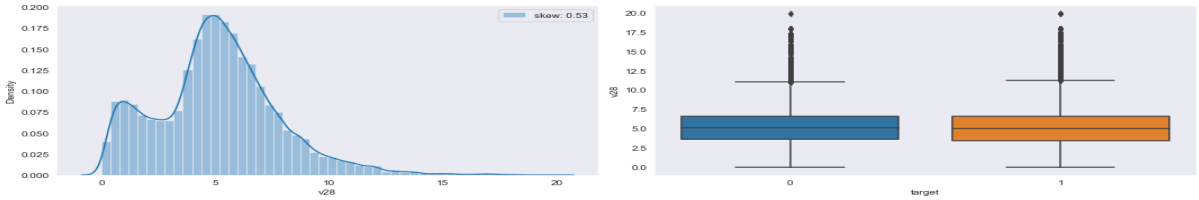
5-number statistics values of v27:

v27 Min: -9.915985826e-07 Q1: 2.19697035586 Q2: 2.6733213970099996 Q3: 3.2243675272900005 Max: 20.00000029

Conclusion: The distribution of column v27 is highly positive/right skewed. Whereas in boxplot a very huge number of outliers (both[q1-1.5*IQR] & [q3+1.5*IQR]) present in v27 with higher % at MAX points w.r.t '0' and '1' case.

Col_name:v28

Pyplot representation of v28:



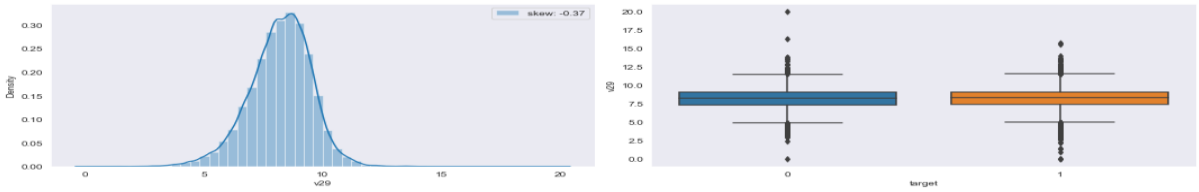
5-number statistics values of v28:

v28 Min: -6.96088017571e-07 Q1: 3.49155347981 Q2: 5.0438313399300005 Q3: 6.5747270702 Max: 19.8481942126

Conclusion: The distribution of column v28 is slightly negative/left skewed. Whereas in boxplot a large number of outliers ([q3+1.5*IQR] region only) present in v28 w.r.t both '0' and '1' case.

Col_name:v29

Pyplot representation of v29:



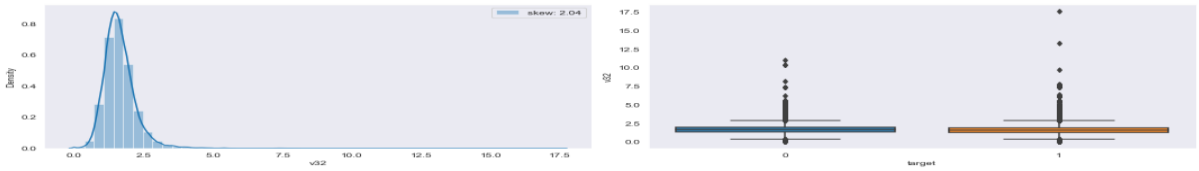
5-number statistics values of v29:

v29 Min: -3.04075268346e-07 Q1: 7.427309980439999 Q2: 8.29613863785 Q3: 9.08786698171 Max: 19.999999918900002

Conclusion: The distribution of column v29 is highly negative/left skewed. Whereas in boxplot a large number of outliers (both[q1-1.5*IQR] & [q3+1.5*IQR]) present in v29 w.r.t both '0' and '1' case.

Col_name:v32

Pyplot representation of v32:



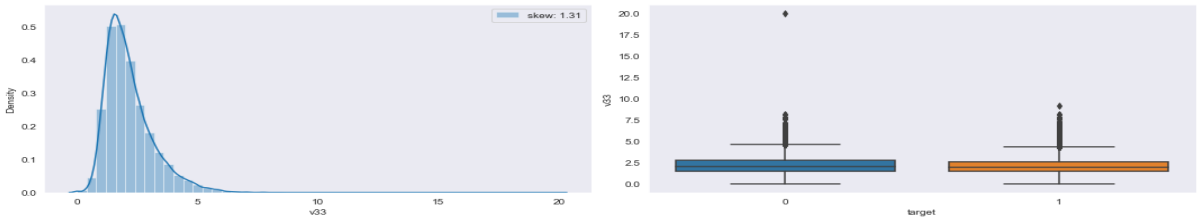
5-number statistics values of v32:

v32 Min: -9.55996033362e-07 Q1: 1.2629539828799998 Q2: 1.5572028434099998 Q3: 1.89632080382 Max: 17.560975144500002

Conclusion: The distribution of column v32 is quite high positive/right skewed. Whereas in boxplot a very huge number of outliers (both[q1-1.5*IQR] & [q3+1.5*IQR]) present in v32 w.r.t both '0' and '1' case.

Col_name:v33

Pyplot representation of v33:

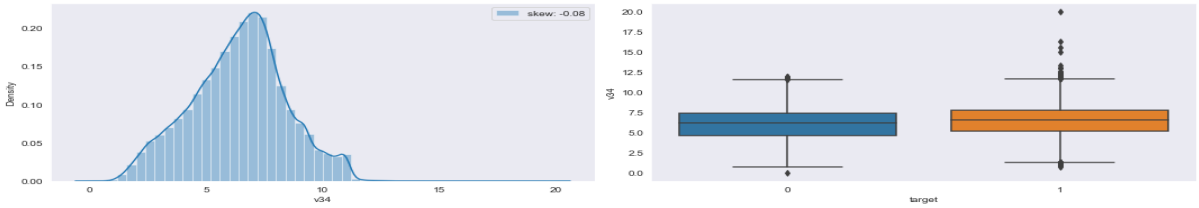


5-number statistics values of v33:

v33 Min: -9.71310804032e-07 Q1: 1.47009594996 Q2: 1.94616943759 Q3: 2.6329112071700003 Max: 20.0000002023
Conclusion: The distribution of column v33 is highly positive/right skewed. Whereas in boxplot a very huge number of outliers ([q3+1.5*IQR] region only) present in v33 w.r.t both '0' and '1' case.

Col_name:v34

Pyplot representation of v34:

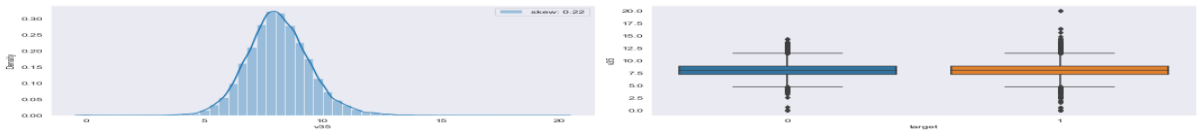


5-number statistics values of v34:

v34 Min: -6.707669798020001e-07 Q1: 5.054164927512501 Q2: 6.537069108995 Q3: 7.702703295347501 Max: 20.000000815
Conclusion: The distribution of column v34 is highly negative/left skewed. Whereas in boxplot a very huge number of outliers (both[q1-1.5*IQR] & [q3+1.5*IQR]) present in v34 w.r.t '1' case low in case '0'.

Col_name:v35

Pyplot representation of v35:

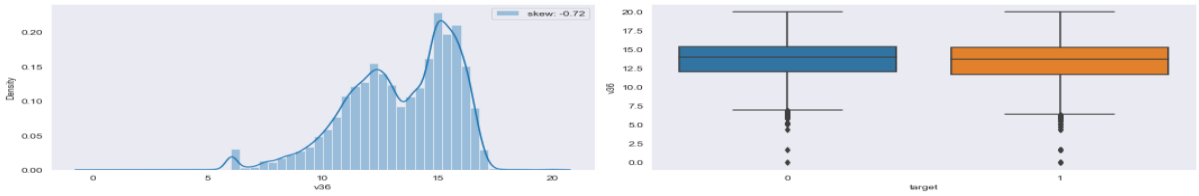


5-number statistics values of v35:

v35 Min: -9.958326503870002e-07 Q1: 7.25388507339 Q2: 8.068505914480001 Q3: 8.93635875049 Max: 20.0000000877
Conclusion: The distribution of column v35 is highly positive/right skewed. Whereas in boxplot a very huge number of outliers (both[q1-1.5*IQR] & [q3+1.5*IQR]) present in v35 w.r.t '1' case low in case '0'.

Col_name:v36

Pyplot representation of v36:

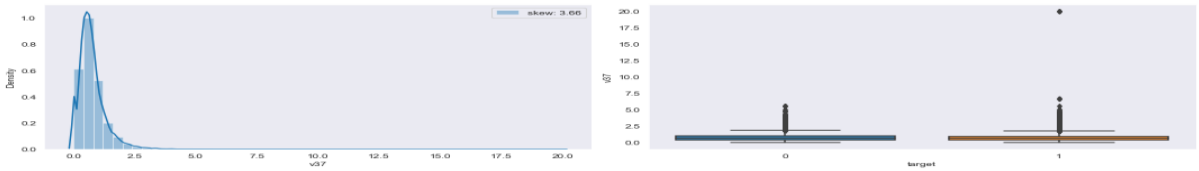


5-number statistics values of v36:

v36 Min: -4.90662773137e-07 Q1: 11.7668064106 Q2: 13.7656165015 Q3: 15.32152941 Max: 20.0000002071
Conclusion: The distribution of column v36 is highly negative/left skewed. Whereas in boxplot a very huge number of outliers ([q1-1.5*IQR] region only) present in v36 w.r.t '0' case and low in '1' case.

Col_name:v37

Pyplot representation of v37:

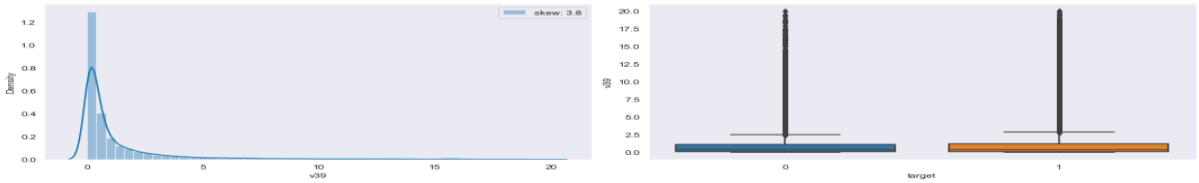


5-number statistics values of v37:

v37 Min: -9.99949815328e-07 Q1: 0.40404126962924997 Q2: 0.64285647783 Q3: 0.9482752200925 Max: 20.000000355
Conclusion: The distribution of column v37 is highly positive/right skewed. Whereas in boxplot a very huge number of outliers ([q3+1.5*IQR] region only) present in v37 with higher % at MAX points w.r.t both '0' an d '1' case.

Col_name:v39

Pyplot representation of v39:

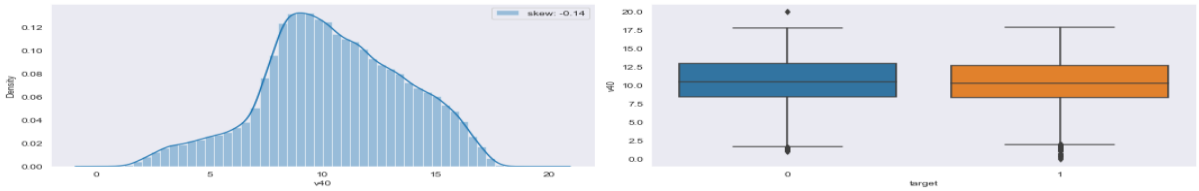


5-number statistics values of v39:

v39 Min: -9.99974223207e-07 Q1: 0.125264068038 Q2: 0.378418331003 Q3: 1.1928299919499998 Max: 19.9155262925
Conclusion: The distribution of column v39 is very highly positive/right skewed. Whereas in boxplot a very huge number of outliers ([q3+1.5*IQR] region only) present in v39 with higher % at MAX points w.r.t both ' 0' and '1' case.

Col name:v40

Pyplot representation of v40:

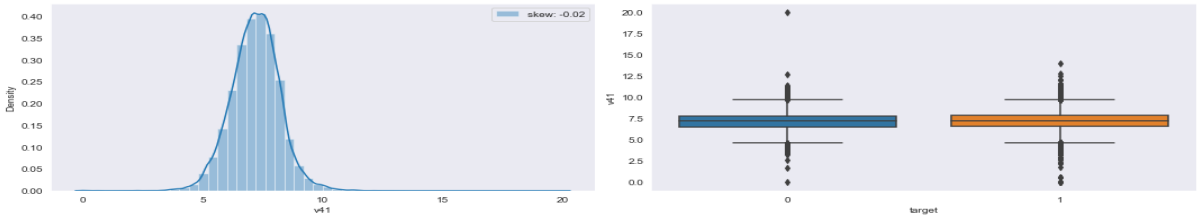


5-number statistics values of v40:

v40 Min: 1.23899595877e-07 Q1: 8.4082200495675 Q2: 10.33427474605 Q3: 12.7651758027 Max: 19.9999991256
Conclusion: The distribution of column v40 is highly positive/right skewed. Whereas in boxplot a very small number of outliers ([q1-1.5*IQR] region only) present in v40 w.r.t '0' and '1' case.

Col name:v41

Pyplot representation of v41:

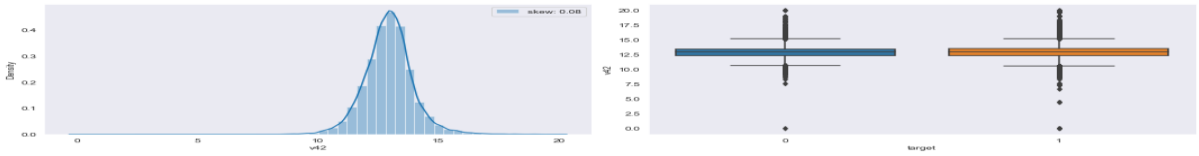


5-number statistics values of v41:

v41 Min: -7.27227521819e-07 Q1: 6.54264977648 Q2: 7.203020350389999 Q3: 7.827246210610001 Max: 19.999999255
Conclusion: The distribution of column v41 is highly negative/left skewed. Whereas in boxplot a large num ber of outliers (both[q1-1.5*IQR] & [q3+1.5*IQR]) present in v41 with higher % at MIN points w.r.t both '0' and '1' case.

Col name:v42

Pyplot representation of v42:

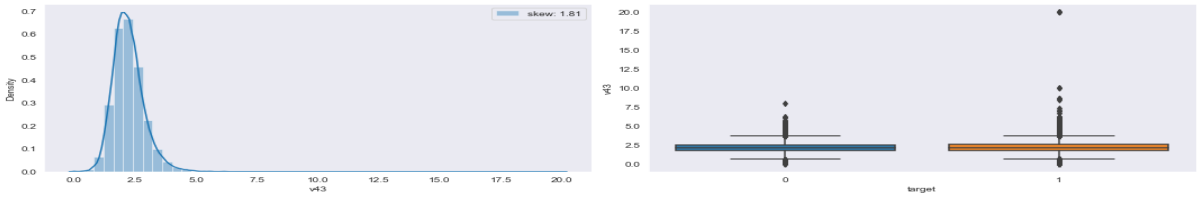


5-number statistics values of v42:

v42 Min: -6.20614377063e-07 Q1: 12.341679176 Q2: 12.9343630952 Q3: 13.4938520107 Max: 19.999999748
Conclusion: The distribution of column v42 is slightly negative/left skewed. Whereas in boxplot a very huge number of outliers (both[q1-1.5*IQR] & [q3+1.5*IQR]) present in v42 w.r.t both '0' and '1' case.

Col_name: v43

Pyplot representation of v43:

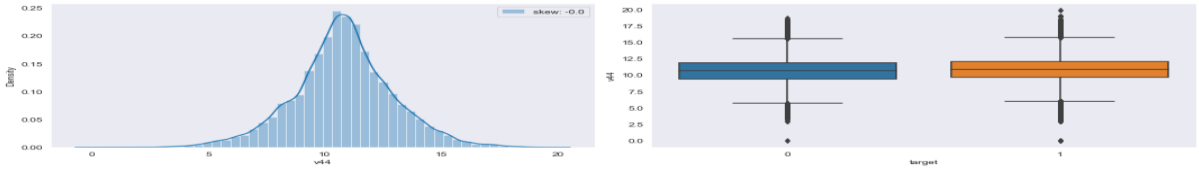


5-number statistics values of v43:

v43 Min: -9.72429495565e-07 Q1: 1.78799554133 Q2: 2.15200715646 Q3: 2.5606786710900002 Max: 20.0000004033
Conclusion: The distribution of column v43 is highly positive/right skewed. Whereas in boxplot a very huge number of outliers (both[q1-1.5*IQR] & [q3+1.5*IQR]) present in v43 w.r.t '1' case compared to low in '0' case.

Col_name:v44

Pyplot representation of v44:

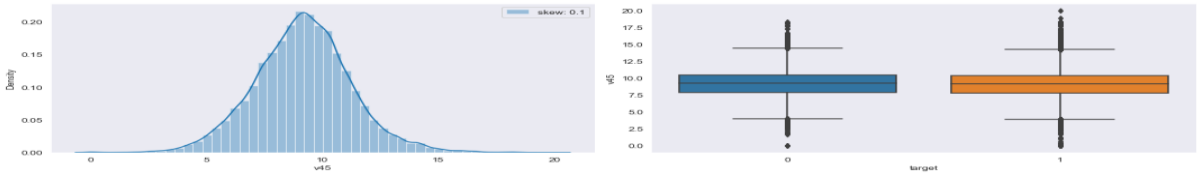


5-number statistics values of v44:

v44 Min: -9.482211927659999e-07 Q1: 9.575103513 Q2: 10.7820081731 Q3: 12.024762221800001 Max: 19.8316812461
Conclusion: The distribution of column v44 is slightly positive/right skewed. Whereas in boxplot a large number of outliers (both[q1-1.5*IQR] & [q3+1.5*IQR]) present in v44 w.r.t '0' and '1' case.

Col_name:v45

Pyplot representation of v45:

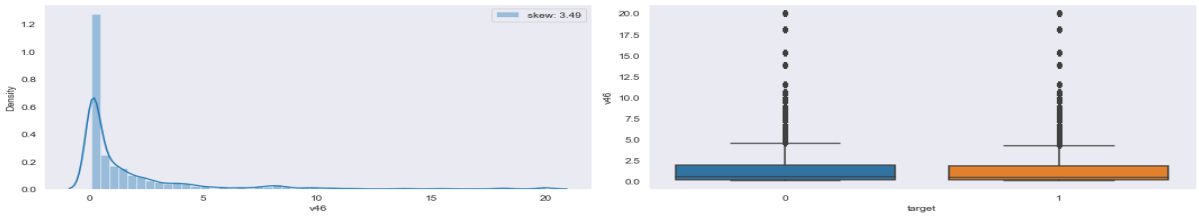


5-number statistics values of v45:

v45 Min: -9.202111848120001e-07 Q1: 7.82664984145 Q2: 9.15604571457 Q3: 10.416813274299999 Max: 20.000000551099998
Conclusion: The distribution of column v45 is quite high negative/left skewed. Whereas in boxplot a very huge number of outliers (both[q1-1.5*IQR] & [q3+1.5*IQR]) present in v45 w.r.t both '0' and '1' case.

Col_name:v46

Pyplot representation of v46:



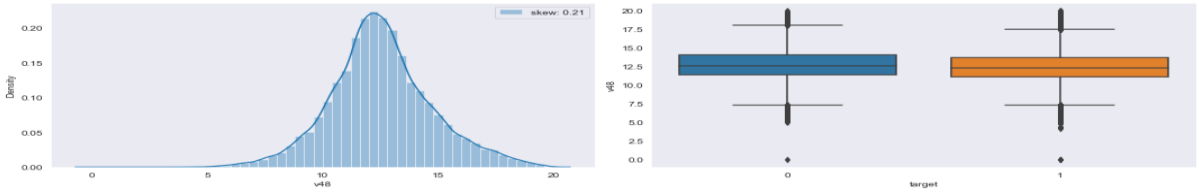
5-number statistics values of v46:

v46 Min: 0.0693487886361 Q1: 0.12370720060024999 Q2: 0.440208836941 Q3: 1.8178720067249998 Max: 20.0000009953

Conclusion: The distribution of column v46 is highly positive/right skewed. Whereas in boxplot a very huge number of outliers ([q3+1.5*IQR] region only) present in v46 with higher % at MAX points w.r.t both '0' and '1' case.

Col_name:v48

Pyplot representation of v48:



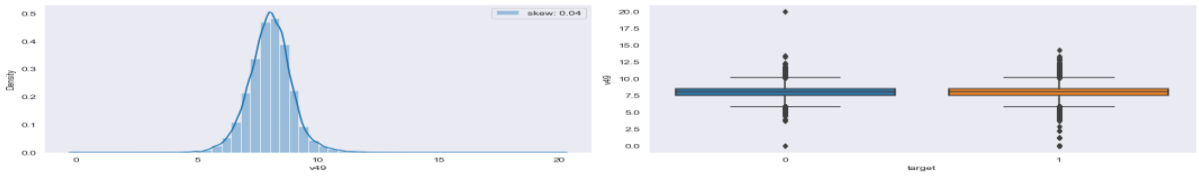
5-number statistics values of v48:

v48 Min: -9.92442155861e-07 Q1: 11.2162474507 Q2: 12.411607858800002 Q3: 13.7779822952 Max: 19.9999996424

Conclusion: The distribution of column v48 is quite high positive/right skewed. Whereas in boxplot a large number of outliers (both[q1-1.5*IQR] & [q3+1.5*IQR]) present in v48 with higher % at MIN points w.r.t both '0' and '1' case.

Col_name:v49

Pyplot representation of v49:



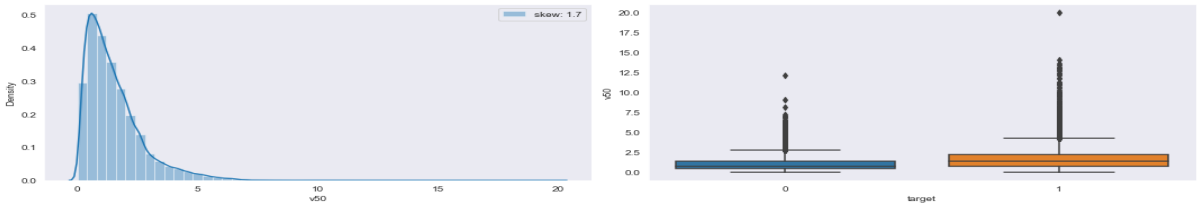
5-number statistics values of v49:

v49 Min: -6.69797512814e-07 Q1: 7.47295890286 Q2: 8.022771306980001 Q3: 8.558951310980001 Max: 19.9999995929

Conclusion: The distribution of column v49 is slightly negative/left skewed. Whereas in boxplot a very huge number of outliers (both[q1-1.5*IQR] & [q3+1.5*IQR]) present in v49 w.r.t both '0' and '1' case.

Col_name:v50

Pyplot representation of v50:



5-number statistics values of v50:

v50 Min: -9.09139305327e-07 Q1: 0.6587917993705 Q2: 1.21194350822 Q3: 2.007188573385 Max: 19.9999991454

Conclusion: The distribution of column v50 is highly positive/right skewed. Whereas in boxplot a very huge number of outliers (both[q1-1.5*IQR] & [q3+1.5*IQR]) present in v50 with higher % at MAX points w.r.t '1' case compared to low in '0'case.

Col_name:v51

Pyplot representation of v51:

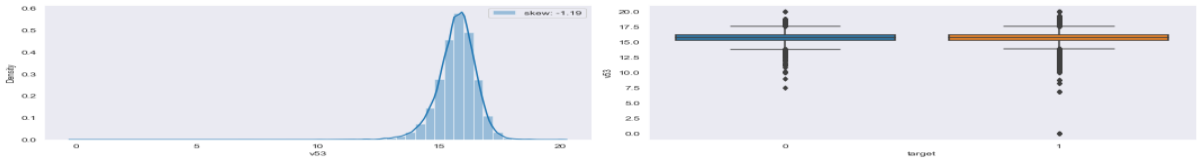
5-number statistics values of v51:

v51 Min: -3.6161218719099996e-07 Q1: 5.59763789643 Q2: 7.134017671989999 Q3: 8.643772014115 Max: 20.00000085 8099998

Conclusion: The distribution of column v51 is quite high negative/left skewed. Whereas in boxplot a large n umber of outliers (both[q1-1.5*IQR] & [q3+1.5*IQR]) present in v51 with higher % at MAX points w.r.t both '0' and '1' case.

Col_name:v53

Pyplot representation of v53:



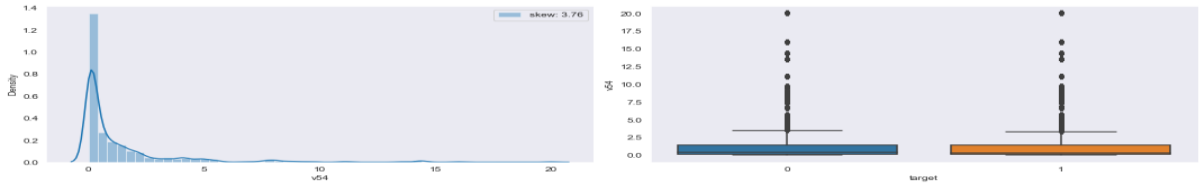
5-number statistics values of v53:

v53 Min: -9.83810668416e-07 Q1: 15.275397366300002 Q2: 15.7709259854 Q3: 16.216867030699998 Max: 20.00000092 2799998

Conclusion: The distribution of column v53 is highly negative/left skewed. Whereas in boxplot a very huge number of outliers (both[q1-1.5*IQR] & [q3+1.5*IQR]) present in v53 with higher % at MIN points w.r.t bot h '0' and '1' case.

Col_name:v54

Pyplot representation of v54:



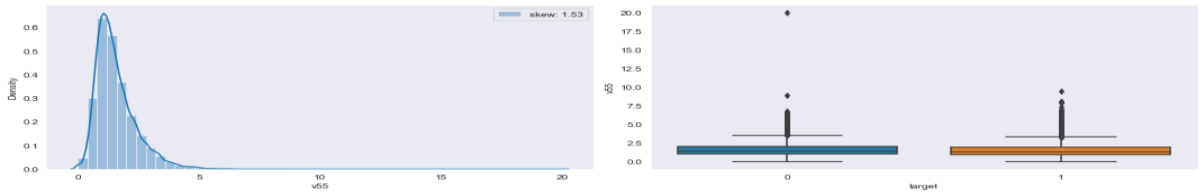
5-number statistics values of v54:

v54 Min: 0.0130615024568 Q1: 0.09309359861005001 Q2: 0.31356516968799997 Q3: 1.4060199791749999 Max: 20.0000009954

Conclusion: The distribution of column v54 is highly positive/right skewed. Whereas in boxplot a very huge number of outliers (both[q1-1.5*IQR] & [q3+1.5*IQR]) present in v54 with higher % at MAX points w.r.t both '0' and '1' case.

Col_name:v55

Pyplot representation of v55:



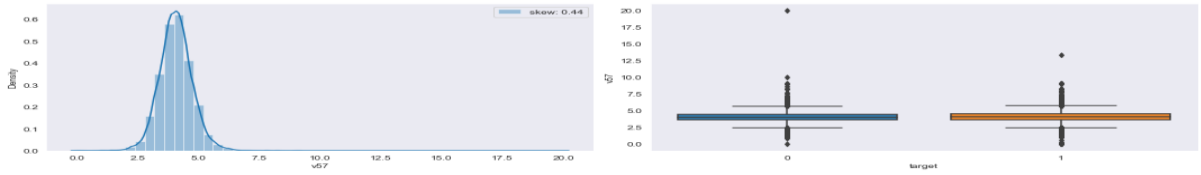
5-number statistics values of v55:

v55 Min: -9.84682005852e-07 Q1: 0.981778329138 Q2: 1.370939696 Q3: 1.94444522572 Max: 20.0000001945

Conclusion: The distribution of column v55 is highly positive/right skewed. Whereas in boxplot a large number of outliers (both[q1-1.5*IQR] & [q3+1.5*IQR]) present in v55 with higher % at MAX points w.r.t both '0' and '1' case.

Col_name:v57

Pyplot representation of v57:



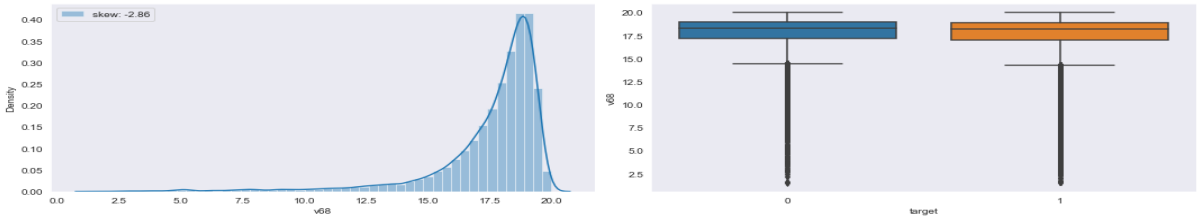
5-number statistics values of v57:

v57 Min: -7.57060718115e-07 Q1: 3.64889244951 Q2: 4.06703923372 Q3: 4.4880550046900005 Max: 19.9999997458

Conclusion: The distribution of column v57 is slightly high positive/ right skewed. Whereas in boxplot a very huge number of outliers (both[q1-1.5*IQR] & [q3+1.5*IQR]) present in v57 w.r.t both '0' and '1' case.

Col_name:v58

Pyplot representation of v58:



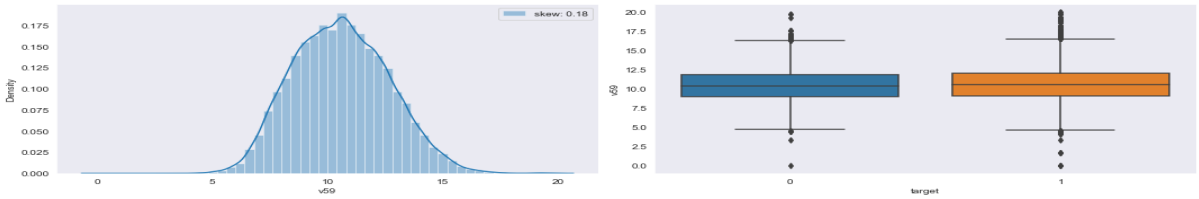
5-number statistics values of v58:

v58 Min: -9.976841401879999e-07 Q1: 1.49999985302 Q2: 5.3305504503500005 Q3: 13.964497957799999 Max: 20.0000009786

Conclusion: The distribution of column v58 is highly positive/right skewed. Whereas in boxplot a very huge number of outliers (both[q1-1.5*IQR] & [q3+1.5*IQR]) present in v58 with higher % at MAX points w.r.t '0' and '1' case.

Col_name:v59

Pyplot representation of v59:



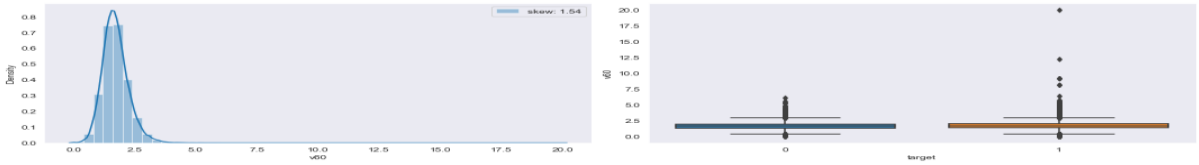
5-number statistics values of v59:

v59 Min: -8.803695286790001e-07 Q1: 9.0557473352 Q2: 10.5351078123 Q3: 12.033088879600001 Max: 19.9999994001

Conclusion: The distribution of column v59 is slightly positive/right skewed. Whereas in boxplot a large number of outliers (both[q1-1.5*IQR] & [q3+1.5*IQR]) present in v59 w.r.t both '0' and '1' case.

Col_name:v60

Pyplot representation of v60:



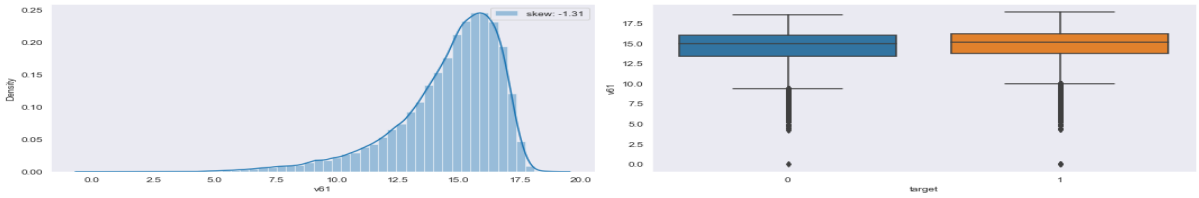
5-number statistics values of v60:

v60 Min: -9.97016404673e-07 Q1: 1.3615257573700001 Q2: 1.6670334485799998 Q3: 2.0105335213599997 Max: 20.000000496600002

Conclusion: The distribution of column v60 is highly positive/right skewed. Whereas in boxplot a very huge number of outliers (both[q1-1.5*IQR] & [q3+1.5*IQR]) present in v60 with higher % at MAX points w.r.t both '0' and '1' case.

Col_name:v61

Pyplot representation of v61:

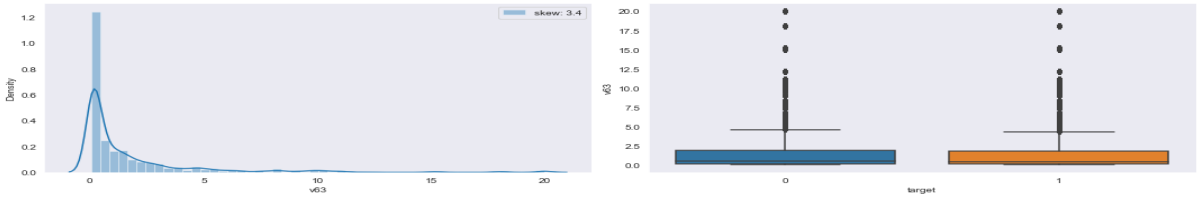


5-number statistics values of v61:

v61 Min: -6.11342815231e-07 Q1: 13.6048103241 Q2: 15.0758945403 Q3: 16.1074846319 Max: 18.8469601202
Conclusion: The distribution of column v61 is highly negative/left skewed. Whereas in boxplot a very huge number of outliers ([q1-1.5*IQR] region only) present in v61 with higher % at MIN points w.r.t both '0' and '1' case.

Col_name:v63

Pyplot representation of v63:

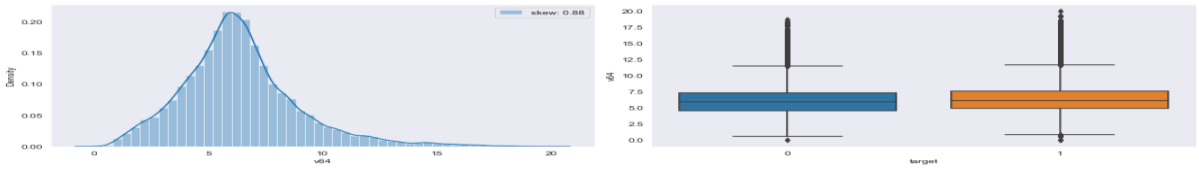


5-number statistics values of v63:

v63 Min: 0.053055275196 Q1: 0.142445490723 Q2: 0.4608322176175 Q3: 1.8489410183275 Max: 20.0000009978
Conclusion: The distribution of column v63 is highly positive/right skewed. Whereas in boxplot a very huge number of outliers ([q3+1.5*IQR] region only) present in v63 w.r.t both '0' and '1' case.

Col_name:v64

Pyplot representation of v64:

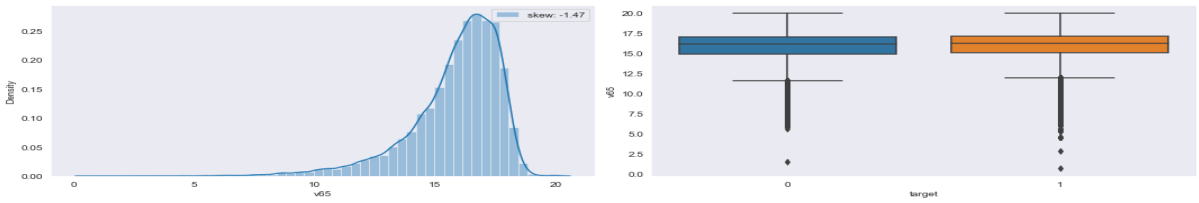


5-number statistics values of v64:

v64 Min: -8.770837351370001e-07 Q1: 4.78521831276 Q2: 6.1088997033700005 Q3: 7.51815951875 Max: 19.9999990118
Conclusion: The distribution of column v64 is highly positive/right skewed. Whereas in boxplot a very huge number of outliers ([q3+1.5*IQR] region only) present in v64 w.r.t both '0' and '1' case.

Col_name:v65

Pyplot representation of v65:

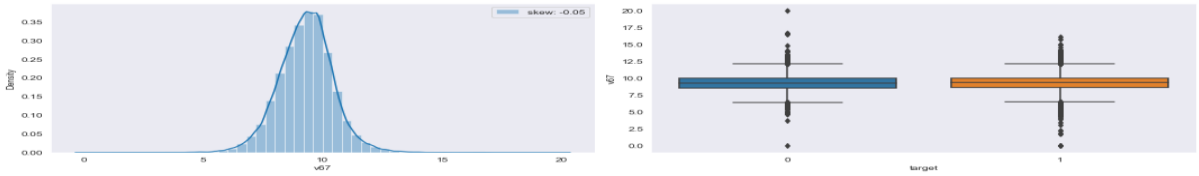


5-number statistics values of v65:

v65 Min: 0.659295685457 Q1: 15.033712203299999 Q2: 16.2625248976 Q3: 17.1642529953 Max: 20.0000009687
Conclusion: The distribution of column v65 is highly negative/left skewed. Whereas in boxplot a very huge number of outliers ([q1-1.5*IQR] region only) present in v65 w.r.t both '0' and '1' case.

Col_name:v67

Pyplot representation of v67:

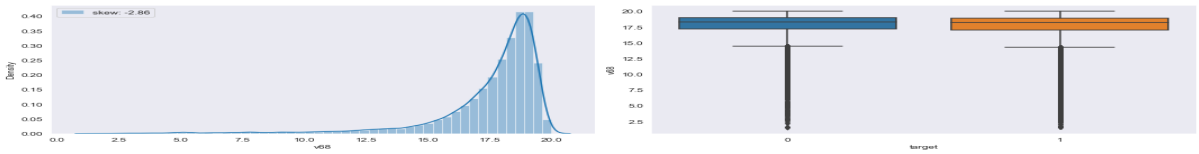


5-number statistics values of v67:

v67 Min: -2.41316145677e-07 Q1: 8.57738317948 Q2: 9.31386090528 Q3: 9.992976825960001 Max: 20.0000009583
Conclusion: The distribution of column v67 is highly negative/left skewed. Whereas in boxplot a very huge number of outliers (both[q1-1.5*IQR] & [q3+1.5*IQR]) present in v67 w.r.t both '0' and '1' case.

Col_name:v68

Pyplot representation of v68:

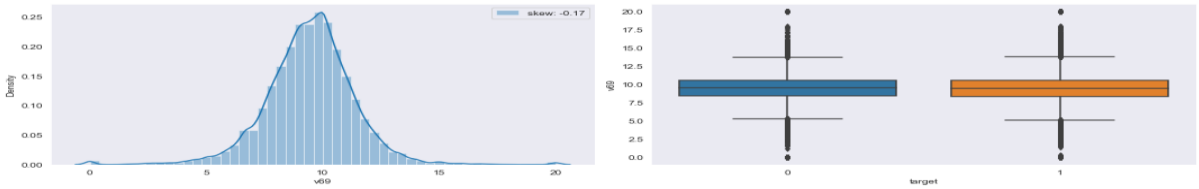


5-number statistics values of v68:

v68 Min: 1.50135913308 Q1: 17.077176516199998 Q2: 18.2745480751 Q3: 18.9113914918 Max: 20.0000009815
Conclusion: The distribution of column v68 is highly negative/left skewed. Whereas in boxplot a very huge number of outliers ([q1-1.5*IQR] region only) present in v68 w.r.t both '0' and '1' case.

Col_name:v69

Pyplot representation of v69:



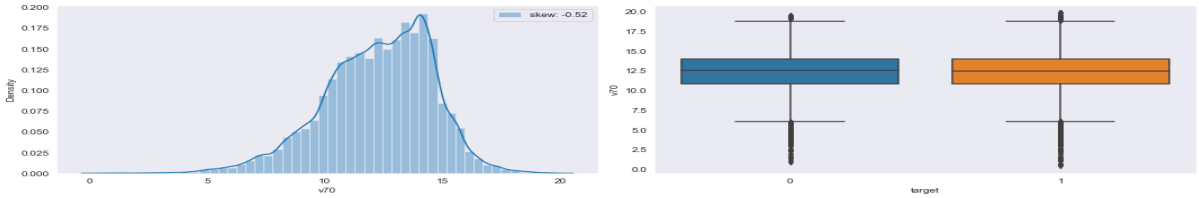
5-number statistics values of v69:

v69 Min: -9.920261848700001e-07 Q1: 8.3898303778575 Q2: 9.51612909794 Q3: 10.537634203125 Max: 20.0000009988

Conclusion: The distribution of column v69 is highly negative/left skewed. Whereas in boxplot a very huge number of outliers (both[q1-1.5*IQR] & [q3+1.5*IQR]) present in v69 w.r.t both '0' and '1' case.

Col_name:v70

Pyplot representation of v70:



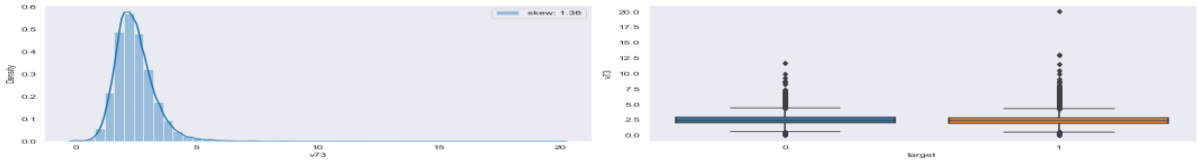
5-number statistics values of v70:

v70 Min: 0.42709459708400005 Q1: 10.802010304100001 Q2: 12.4948723899 Q3: 13.994624274100001 Max: 19.816310918800003

Conclusion: The distribution of column v70 is slightly high negative/left skewed. Whereas in boxplot a very huge number of outliers (both[q1-1.5*IQR] & [q3+1.5*IQR]) present in v70 with higher % at MAX points w.r.t both '0' and '1' case.

Col_name:v73

Pyplot representation of v73:



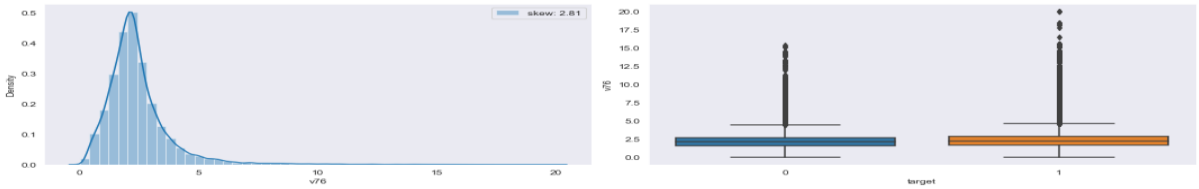
5-number statistics values of v73:

v73 Min: -9.83859158467e-07 Q1: 1.9021729853 Q2: 2.33126130695 Q3: 2.85055794333 Max: 19.999999641800002

Conclusion: The distribution of column v73 is highly positive/right skewed. Whereas in boxplot a very huge number of outliers (both[q1-1.5*IQR] & [q3+1.5*IQR]) present in v73 with higher % at MAX points w.r.t both '0' and '1' case.

Col_name:v76

Pyplot representation of v76:

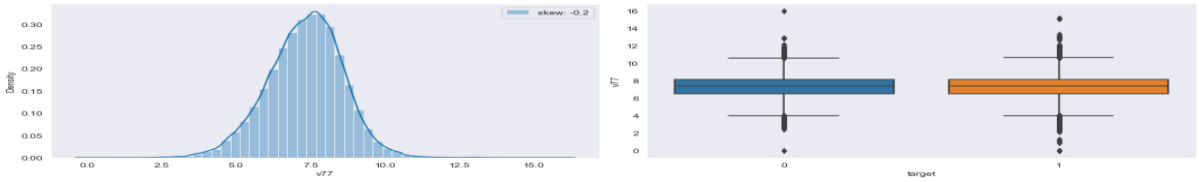


5-number statistics values of v76:

v76 Min: -6.17196681108e-07 Q1: 1.6273841821200001 Q2: 2.17029980441 Q3: 2.80653255959 Max: 20.0000005707
Conclusion: The distribution of column v76 is highly positive/right skewed. Whereas in boxplot a very large number of outliers ([q3+1.5*IQR] region only) present in v76 w.r.t both '0' and '1' case.

Col_name:v77

Pyplot representation of v77:

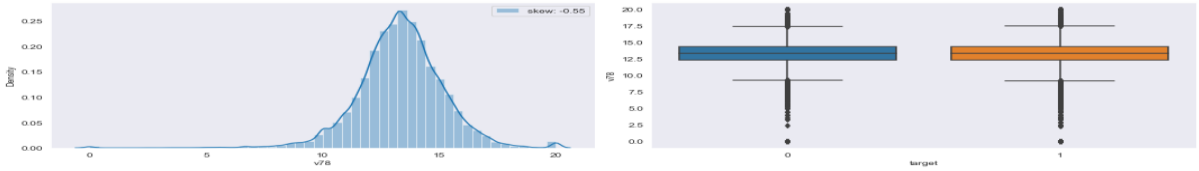


5-number statistics values of v77:

v77 Min: -7.72989617027e-07 Q1: 6.504956688080001 Q2: 7.3758250511000005 Q3: 8.16390087653 Max: 15.973508998
1
Conclusion: The distribution of column v77 is highly negative/left skewed. Whereas in boxplot a very large number of outliers (both[q1-1.5*IQR] & [q3+1.5*IQR]) present in v77 w.r.t both '0' and '1' case.

Col_name:v78

Pyplot representation of v78:

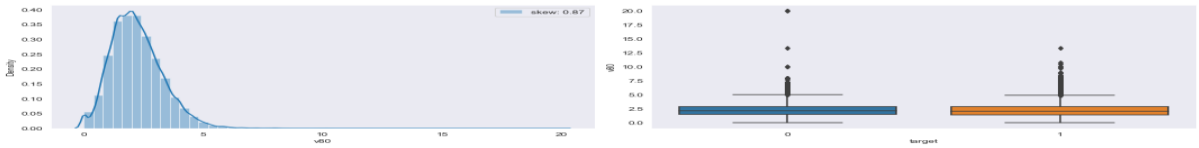


5-number statistics values of v78:

v78 Min: -9.90257208138e-07 Q1: 12.313432547200001 Q2: 13.3333336666 Q3: 14.393938910925 Max: 20.0000009888
Conclusion: The distribution of column v78 is highly positive/right skewed. Whereas in boxplot a very large number of outliers (both[q1-1.5*IQR] & [q3+1.5*IQR]) present in v78 with higher % at MIN points w.r.t bot h '0' and '1' case.

Col_name:v80

Pyplot representation of v80:



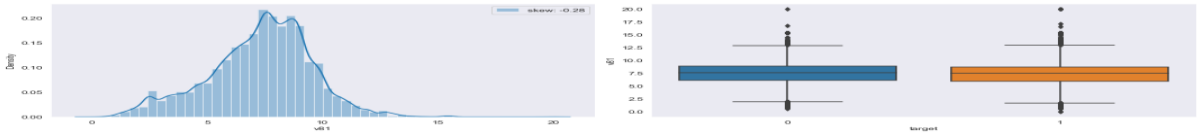
5-number statistics values of v80:

v80 Min: -9.992918915690001e-07 Q1: 1.4465410098175 Q2: 2.09090923567 Q3: 2.85714187893 Max: 19.9999996377

Conclusion: The distribution of column v80 is highly positive/right skewed. Whereas in boxplot a very large number of outliers ([q3+1.5*IQR] region only) present in v80 w.r.t both '0' and '1' case.

Col_name:v81

Pyplot representation of v81:



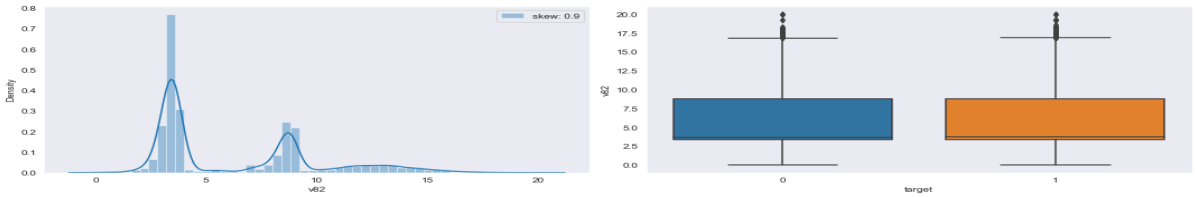
5-number statistics values of v81:

v81 Min: -1.44376504816e-07 Q1: 5.98321242748 Q2: 7.517741523989999 Q3: 8.7760522541 Max: 20.0000009706

Conclusion: The distribution of column v81 is highly negative/left skewed. Whereas in boxplot a very large number of outliers (both[q1-1.5*IQR] & [q3+1.5*IQR]) present in v81 with higher % at MAX points w.r.t bot h '0' and '1' case.

Col_name:v82

Pyplot representation of v82:



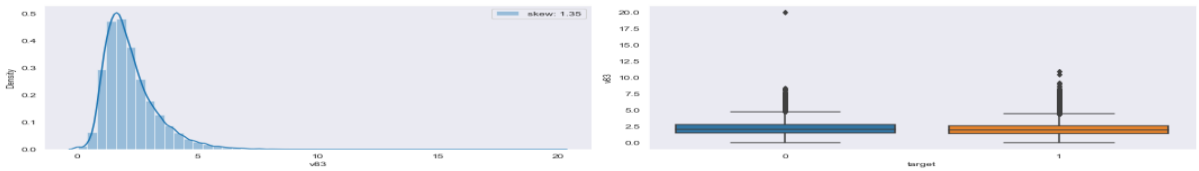
5-number statistics values of v82:

v82 Min: -7.25176672095e-07 Q1: 3.38529988989 Q2: 3.68845932474 Q3: 8.791594287 Max: 20.0000002311

Conclusion: The distribution of column v82 is highly positive/right skewed. Whereas in boxplot few numbe r of outliers ([q3+1.5*IQR] region only) present in v82 w.r.t both '0' and '1' case.

Col_name:v83

Pyplot representation of v83:



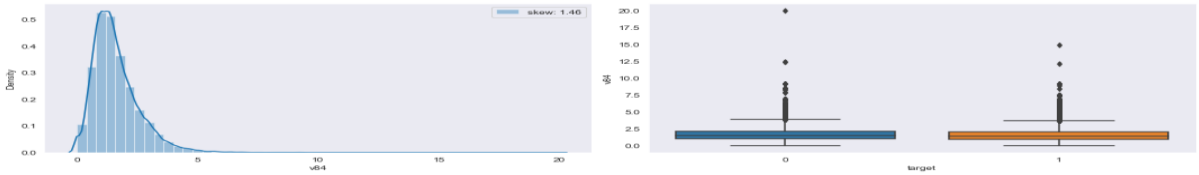
5-number statistics values of v83:

v83 Min: -9.86567916419e-07 Q1: 1.43276110966 Q2: 1.9372439349000001 Q3: 2.66816057928 Max: 20.0000002847

Conclusion: The distribution of column v83 is highly positive/right skewed. Whereas in boxplot a large number of outliers ([q3+1.5*IQR] region only) present in v83 w.r.t both '0' and '1' case.

Col_name:v84

Pyplot representation of v84:



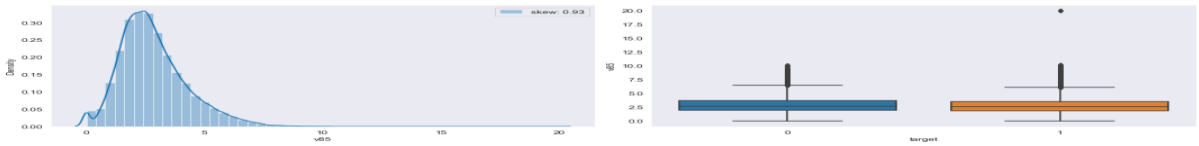
5-number statistics values of v84:

v84 Min: -9.994244670979998e-07 Q1: 0.956599690316 Q2: 1.4232945100499999 Q3: 2.06945390743 Max: 20.000000969 9

Conclusion: The distribution of column v84 is highly / skewed. Whereas in boxplot a very huge number of outliers (both [q1-1.5*IQR] & [q3+1.5*IQR]) present in v84 with w.r.t both '0' and '1' case.

Col_name:v85

Pyplot representation of v85:



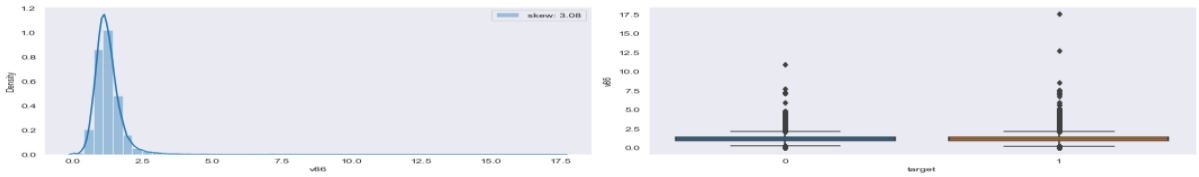
5-number statistics values of v85:

v85 Min: -9.9957405643e-07 Q1: 1.83673522587 Q2: 2.5949368588099997 Q3: 3.5744684134150004 Max: 20.000000888 9

Conclusion: The distribution of column v85 is highly / skewed. Whereas in boxplot a very huge number of outliers (both[q1-1.5*IQR] & [q3+1.5*IQR]) present in v85 with higher % at MAX points w.r.t '0' and '1' case.

Col_name:v86

Pyplot representation of v86:



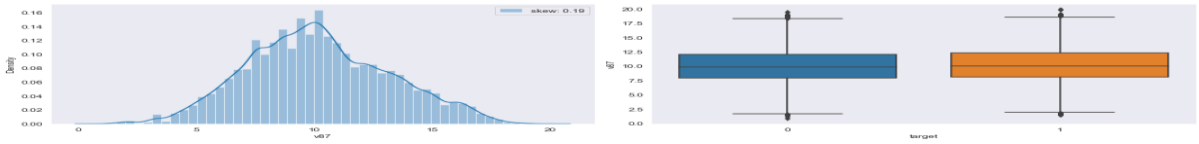
5-number statistics values of v86:

v86 Min: -9.81430998611e-07 Q1: 0.9370127669830001 Q2: 1.15830099984 Q3: 1.4220508048500002 Max: 17.5609751085

Conclusion: The distribution of column v86 is highly positive/right skewed. Whereas in boxplot a very large number of outliers (both[q1-1.5*IQR] & [q3+1.5*IQR]) present in v86 with higher % at MAX points w.r.t both '0' and '1' case.

Col_name:v87

Pyplot representation of v87:



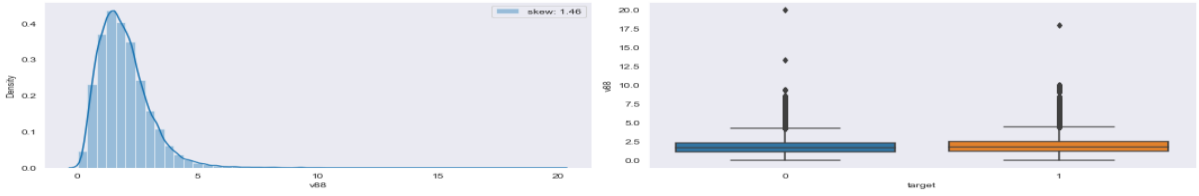
5-number statistics values of v87:

v87 Min: 0.872395517079 Q1: 8.05542532308 Q2: 9.995718014605 Q3: 12.2304772709 Max: 19.8427544339

Conclusion: The distribution of column v87 is highly positive/right skewed. Whereas in boxplot few number of outliers (both[q1-1.5*IQR] & [q3+1.5*IQR]) present in v87 w.r.t both '0' and '1' case.

Col_name:v88

Pyplot representation of v88:



5-number statistics values of v88:

v88 Min: -9.99008151668e-07 Q1: 1.18027583566 Q2: 1.7615474857900002 Q3: 2.4615388884700002 Max: 20.0000000942

Conclusion: The distribution of column v88 is highly positive/right skewed. Whereas in boxplot a large number of outliers (both[q1-1.5*IQR] & [q3+1.5*IQR]) present in v88 with higher % at MAX points w.r.t both '0' and '1' case.

Col_name:v89

Pyplot representation of v89:

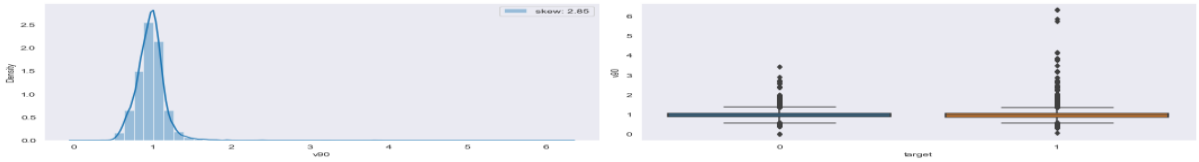
5-number statistics values of v89:

v89 Min: 0.022365182965199997 Q1: 0.099645086547375 Q2: 0.332364022714 Q3: 1.74606431997 Max: 20.0000009972

Conclusion: The distribution of column v89 is highly positive/right skewed. Whereas in boxplot a very large number of outliers (both[q1-1.5*IQR] & [q3+1.5*IQR]) present in v89 with higher % at MAX points w.r.t both ‘0’ and ‘1’ case.

Col_name:v90

Pyplot representation of v90:



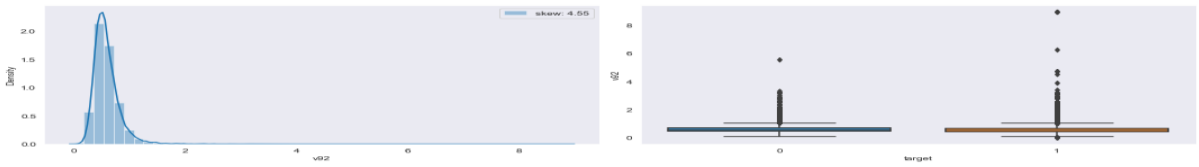
5-number statistics values of v90:

v90 Min: -2.3430042146399998e-07 Q1: 0.8627289394230001 Q2: 0.969182916547 Q3: 1.0617790735400001 Max: 6.305 77492863

Conclusion: The distribution of column v90 is highly negative/left skewed. Whereas in boxplot a very large number of outliers (both[q1-1.5*IQR] & [q3+1.5*IQR]) present in v90 with higher % at MAX points w.r.t both ‘0’ and ‘1’ case.

Col_name:v92

Pyplot representation of v92:



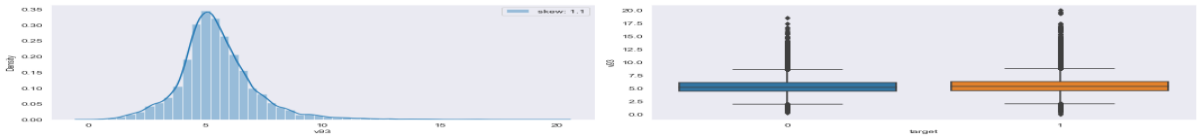
5-number statistics values of v92:

v92 Min: 3.3932097477700003e-08 Q1: 0.43504684494025003 Q2: 0.5426693512630001 Q3: 0.6783547013669999 Max: 8.92384346581

Conclusion: The distribution of column v92 is highly positive/right skewed. Whereas in boxplot a very large number of outliers ([q3+1.5*IQR] region only) present in v92 with higher % at MAX points w.r.t both ‘0’ and ‘1’ case.

Col_name:v93

Pyplot representation of v93:



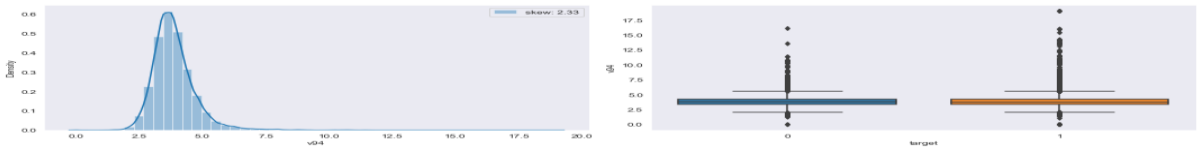
5-number statistics values of v93:

v93 Min: 4.2519186131599997e-07 Q1: 4.548473352469999 Q2: 5.30104695254 Q3: 6.223883498919999 Max: 19.9999994839

Conclusion: The distribution of column v93 is highly positive/right skewed. Whereas in boxplot a very large number of outliers (both[q1-1.5*IQR] & [q3+1.5*IQR]) present in v93 with higher % at MAX points w.r.t both '0' and '1' case.

Col_name:v94

Pyplot representation of v94:



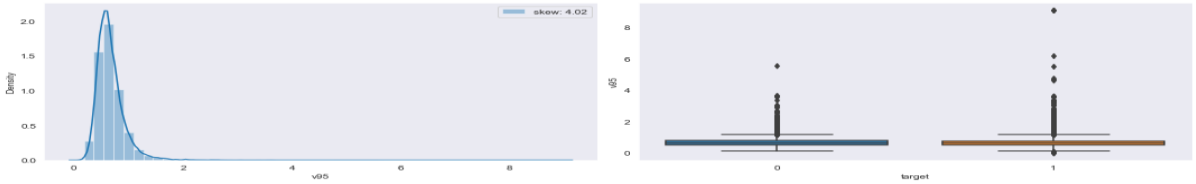
5-number statistics values of v94:

v94 Min: -9.687808680459998e-07 Q1: 3.3339318784599996 Q2: 3.7431063708999996 Q3: 4.22506276561 Max: 19.0163118283

Conclusion: The distribution of column v94 is highly positive/right skewed. Whereas in boxplot a very large number of outliers (both[q1-1.5*IQR] & [q3+1.5*IQR]) present in v94 with higher % at MAX points w.r.t both '0' and '1' case.

Col_name:v95

Pyplot representation of v95:



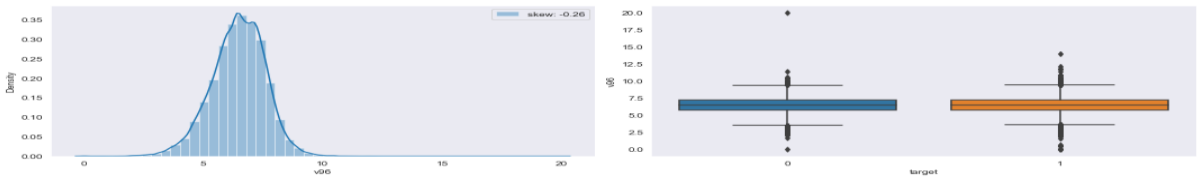
5-number statistics values of v95:

v95 Min: -5.78467359178e-07 Q1: 0.5016116954102501 Q2: 0.6201132443724999 Q3: 0.7691489946582499 Max: 9.070537703200001

Conclusion: The distribution of column v95 is highly positive/right skewed. Whereas in boxplot a very large number of outliers (both[q1-1.5*IQR] & [q3+1.5*IQR]) present in v95 with higher % at MAX points w.r.t both '0' and '1' case.

Col_name:v96

Pyplot representation of v96:



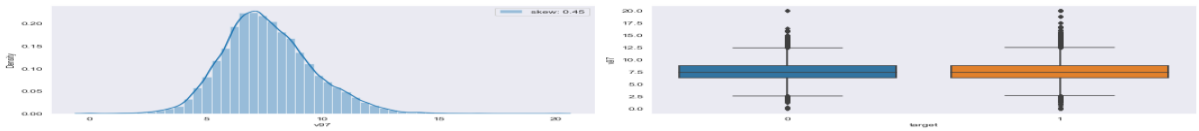
5-number statistics values of v96:

v96 Min: -8.83250412885e-07 Q1: 5.758412042430001 Q2: 6.5116270730900006 Q3: 7.23076870288 Max: 19.9999997693

Conclusion: The distribution of column v96 is quite high negative/left skewed. Whereas in boxplot a very large number of outliers (both[q1-1.5*IQR] & [q3+1.5*IQR]) present in v96 w.r.t both '0' and '1' case.

Col_name:v97

Pyplot representation of v97:



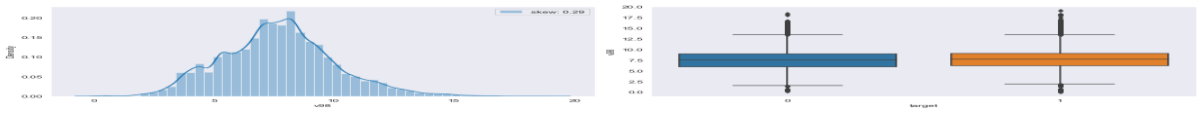
5-number statistics values of v97:

v97 Min: -9.059520411790001e-07 Q1: 6.315790173055 Q2: 7.446807548015 Q3: 8.780488305797501 Max: 20.000000833199998

Conclusion: The distribution of column v97 is highly positive/right skewed. Whereas in boxplot a very large number of outliers (both[q1-1.5*IQR] & [q3+1.5*IQR]) present in v97 w.r.t both '0' and '1' case.

Col_name:v98

Pyplot representation of v98:



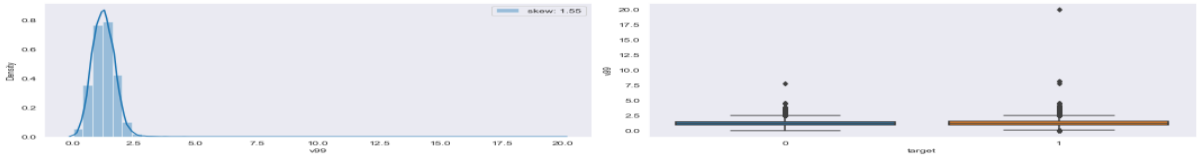
5-number statistics values of v98:

v98 Min: -5.60592762574e-07 Q1: 6.12528780775 Q2: 7.644154053589999 Q3: 9.064070021974999 Max: 19.0587996349

Conclusion: The distribution of column v98 is highly negative/left skewed. Whereas in boxplot a very large number of outliers (both[q1-1.5*IQR] & [q3+1.5*IQR]) present in v98 w.r.t both '0' and '1' case.

Col_name:v99

Pyplot representation of v99:



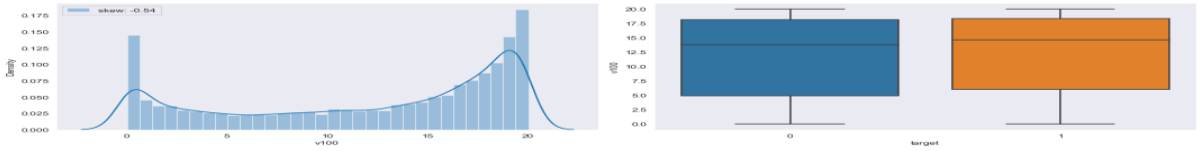
5-number statistics values of v99:

v99 Min: -9.78730918601e-07 Q1: 0.9330819005709999 Q2: 1.2355462178399998 Q3: 1.55250036745 Max: 19.9999997618

Conclusion: The distribution of column v99 is highly positive/right skewed. Whereas in boxplot a very large number of outliers ([q3+1.5*IQR] region only) present in v99 with higher % at MAX points w.r.t both '0' and '1' case.

Col_name:v100

Pyplot representation of v100:



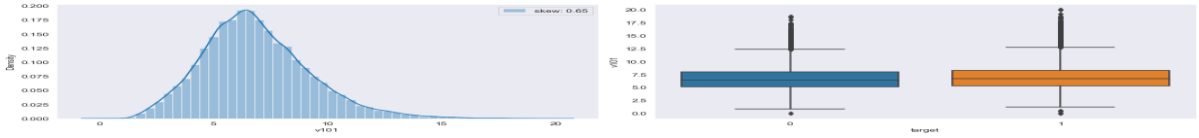
5-number statistics values of v100:

v100 Min: -9.98131133939e-07 Q1: 5.754165593280001 Q2: 14.4759391925 Q3: 18.317460345 Max: 20.0000009983

Conclusion: The distribution of column v100 is very highly negative/left skewed. Whereas in boxplot no outliers (both[q1-1.5*IQR] & [q3+1.5*IQR]) present in v100 w.r.t '0' and '1' case.

Col_name:v101

Pyplot representation of v101:



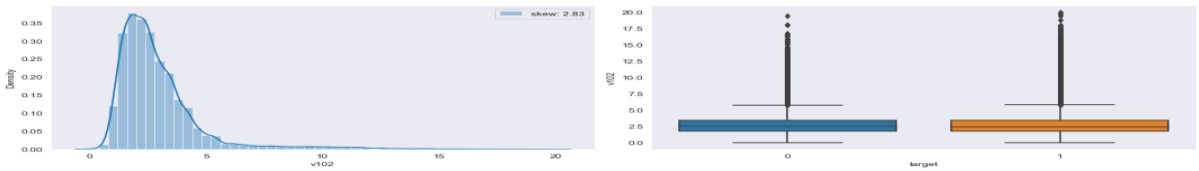
5-number statistics values of v101:

v101 Min: -3.94276655676e-07 Q1: 5.25797252073 Q2: 6.623713610529999 Q3: 8.24079367823 Max: 20.0000004219

Conclusion: The distribution of column v101 is very highly positive/right skewed. Whereas in boxplot a very large number of outliers ([q3+1.5*IQR] region only) present in v101 w.r.t both '0' and '1' case.

Col_name:v102

Pyplot representation of v102:

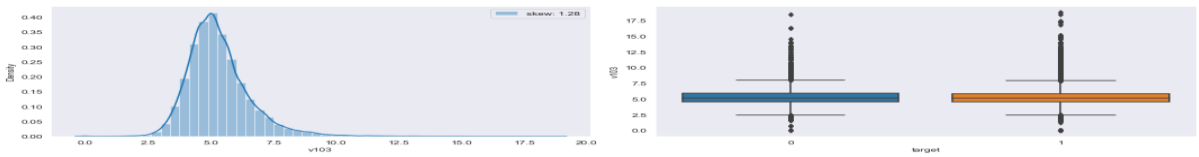


5-number statistics values of v102:

v102 Min: -7.11149257395e-07 Q1: 1.7936322062 Q2: 2.46289829216 Q3: 3.4132556245999996 Max: 20.0000001121
Conclusion: The distribution of column v102 is very highly positive/right skewed. Whereas in boxplot a very large number of outliers ([q3+1.5*IQR] region only) present in v102 w.r.t both '0' and '1' case.

Col_name:v103

Pyplot representation of v103:



5-number statistics values of v103:

v103 Min: -9.75774270091e-07 Q1: 4.50090380363 Q2: 5.12584650255 Q3: 5.87454484789 Max: 18.7752514628
Conclusion: The distribution of column v103 is highly positive/right skewed. Whereas in boxplot a very larg e number of outliers (both[q1-1.5*IQR] & [q3+1.5*IQR]) present in v103 with higher % at MAX points w.r.t both '0' and '1' case.

Col_name:v104

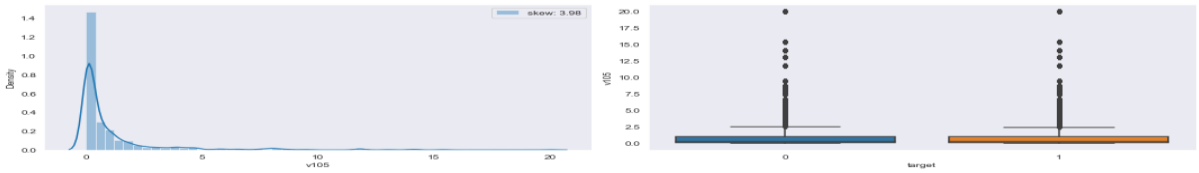
Pyplot representation of v104:

5-number statistics values of v104:

v104 Min: -8.68316931708e-07 Q1: 2.1409315967800002 Q2: 2.5120338385 Q3: 2.9487535970499996 Max: 20.00000060 4500002
Conclusion: The distribution of column v104 is very highly positive/right skewed. Whereas in boxplot a very large number of outliers (both[q1-1.5*IQR] & [q3+1.5*IQR]) present in v104 with higher % at MAX points w .r.t both '0' and '1' case.

Col_name:v105

Pyplot representation of v105:



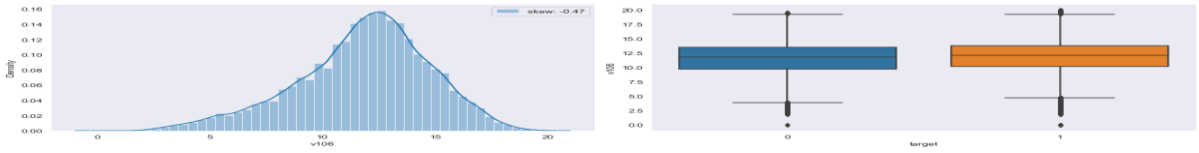
5-number statistics values of v105:

v105 Min: 9.36498075773e-05 Q1: 0.055564883647999996 Q2: 0.24255580969300003 Q3: 1.0209482797849998 Max: 20.0000009841

Conclusion: The distribution of column v105 is very much highly positive/right skewed. Whereas in boxplot a very large number of outliers (both[q1-1.5*IQR] & [q3+1.5*IQR]) present in v105 with higher % at MAX points w.r.t both '0' and '1' case.

Col_name:v106

Pyplot representation of v106:



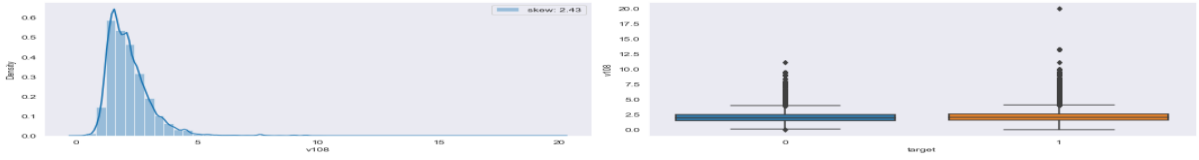
5-number statistics values of v106:

v106 Min: -5.46702886447e-07 Q1: 10.0498086021 Q2: 12.085176797699999 Q3: 13.767514950199999 Max: 20.00000033 58

Conclusion: The distribution of column v106 is highly negative/left skewed. Whereas in boxplot a very large number of outliers (both[q1-1.5*IQR] & [q3+1.5*IQR]) present in v106 with higher % at MIN points w.r.t both '0' and '1' case.

Col_name:v108

Pyplot representation of v108:



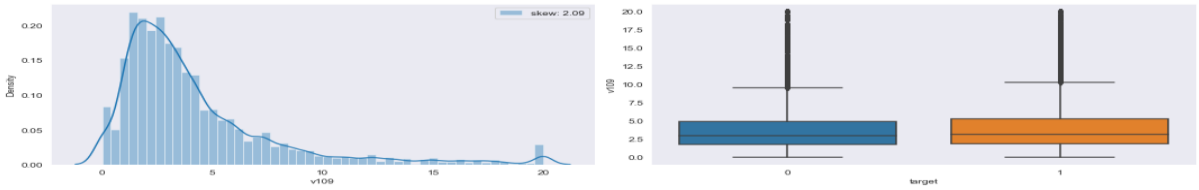
5-number statistics values of v108:

v108 Min: 4.25164336154e-07 Q1: 1.5303668240000001 Q2: 1.9808336402900002 Q3: 2.5416022918400003 Max: 20.0000005938

Conclusion: The distribution of column v108 is very much highly positive/right skewed. Whereas in boxplot a very large number of outliers (both[q1-1.5*IQR] & [q3+1.5*IQR]) present in v108 with higher % at MAX points w.r.t both '0' and '1' case.

Col_name:v109

Pyplot representation of v109:



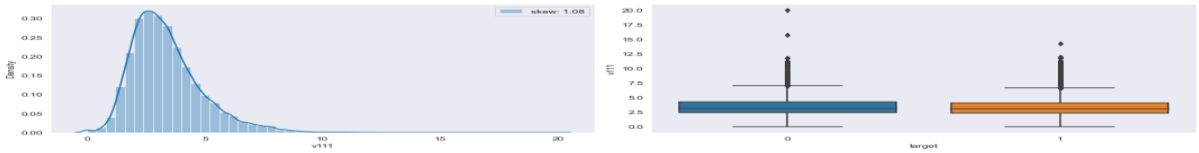
5-number statistics values of v109:

v109 Min: -9.996884417950002e-07 Q1: 1.8421062064400002 Q2: 3.08790872851 Q3: 5.1519411014 Max: 20.0000009996

Conclusion: The distribution of column v109 is highly positive/right skewed. Whereas in boxplot a very large number of outliers ([q3+1.5*IQR] region only) present in v109 w.r.t both '0' and '1' case.

Col_name:v111

Pyplot representation of v111:



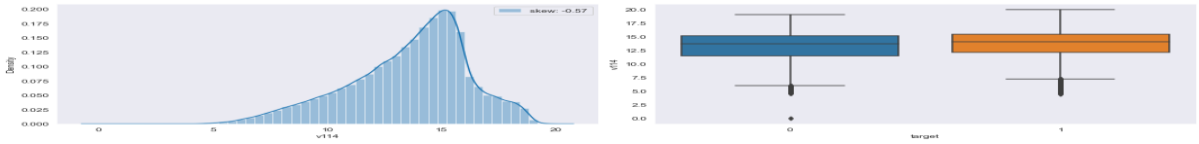
5-number statistics values of v111:

v111 Min: -9.990269535219999e-07 Q1: 2.3175966204 Q2: 3.10880880661 Q3: 4.120879210769999 Max: 20.0000005066

Conclusion: The distribution of column v111 is highly positive/right skewed. Whereas in boxplot a very large number of outliers ([q3+1.5*IQR] region only) present in v111 w.r.t both '0' and '1' case.

Col_name:v114

Pyplot representation of v114:



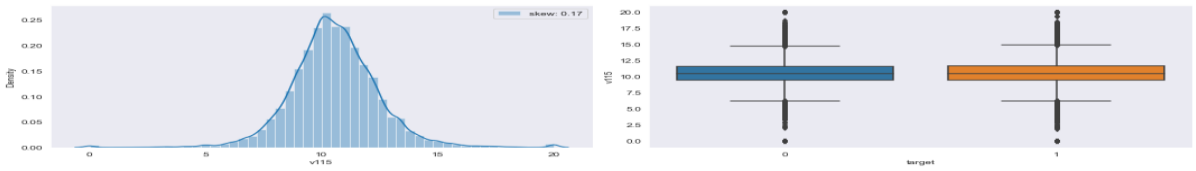
5-number statistics values of v114:

v114 Min: -9.33532697748e-10 Q1: 11.995312033800001 Q2: 14.0399804116 Q3: 15.37266637395 Max: 19.999999675599998

Conclusion: The distribution of column v114 is highly negative/left skewed. Whereas in boxplot few number of outliers ([q1-1.5*IQR] region only) present in v114 w.r.t both '0' and '1' case.

Col_name:v115

Pyplot representation of v115:



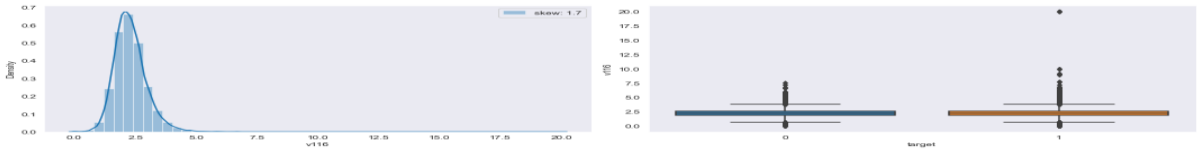
5-number statistics values of v115:

v115 Min: -9.85318869981e-07 Q1: 9.457365023900001 Q2: 10.476190984950001 Q3: 11.612902282249998 Max: 20.0000009715

Conclusion: The distribution of column v115 is highly positive/right skewed. Whereas in boxplot a very large number of outliers (both[q1-1.5*IQR] & [q3+1.5*IQR]) present in v115 w.r.t both '0' and '1' case.

Col_name:v116

Pyplot representation of v116:



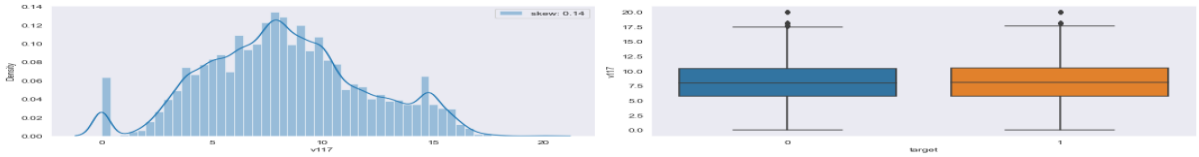
5-number statistics values of v116:

v116 Min: -9.450358758540001e-07 Q1: 1.85147438062 Q2: 2.2222232203599996 Q3: 2.6496808725400003 Max: 20.0000007973

Conclusion: The distribution of column v116 is highly positive/right skewed. Whereas in boxplot a very large number of outliers (both[q1-1.5*IQR] & [q3+1.5*IQR]) present in v116 with higher % at MAX points w.r.t both '0' and '1' case.

Col_name:v117

Pyplot representation of v117:



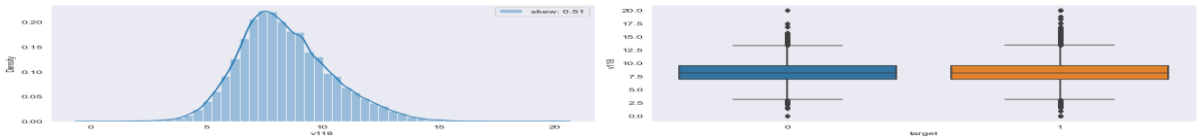
5-number statistics values of v117:

v117 Min: -9.9919923615e-07 Q1: 5.76334479858 Q2: 8.070174962989999 Q3: 10.5032167995 Max: 20.0000007954

Conclusion: The distribution of column v117 is very highly positive/right skewed. Whereas in boxplot negligible number of outliers (both[q1-1.5*IQR] & [q3+1.5*IQR]) present in v117 w.r.t both '0' and '1' case.

Col_name:v118

Pyplot representation of v118:



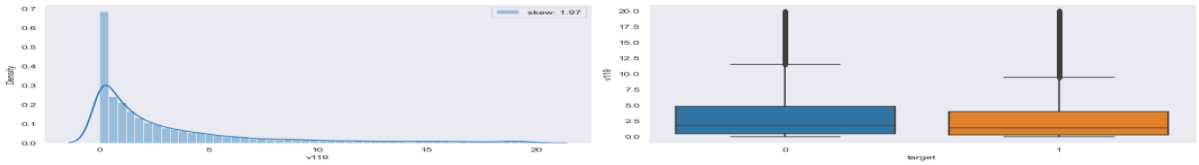
5-number statistics values of v118:

v118 Min: -1.69546317645e-07 Q1: 6.9816274274975 Q2: 8.136963749185 Q3: 9.5652183025475 Max: 20.0000009555

Conclusion: The distribution of column v118 is very highly positive/right skewed. Whereas in boxplot a very large number of outliers (both[q1-1.5*IQR] & [q3+1.5*IQR]) present in v118 with higher % at MAX points w .r.t both '0' and '1' case.

Col_name:v119

Pyplot representation of v119:



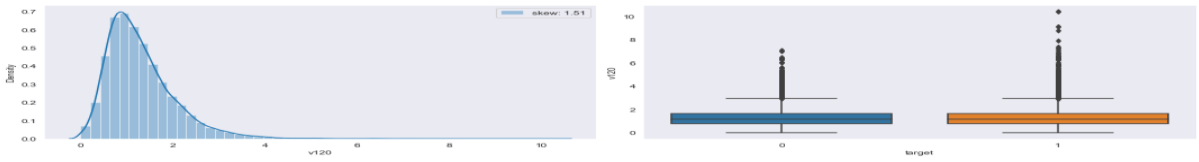
5-number statistics values of v119:

v119 Min: -9.998182866889999e-07 Q1: 0.315008651029 Q2: 1.4615609348700003 Q3: 4.16604880471 Max: 20.0000004599

Conclusion: The distribution of column v119 is very highly positive/right skewed. Whereas in boxplot a very large number of outliers (both[q1-1.5*IQR] & [q3+1.5*IQR]) present in v119 with higher % at MAX points w .r.t both '0' and '1' case.

Col_name:v120

Pyplot representation of v120:



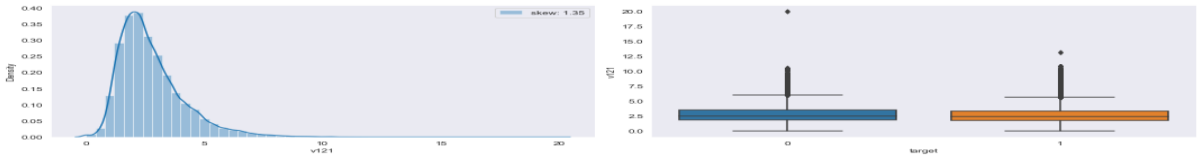
5-number statistics values of v120:

v120 Min: -9.93253443003e-07 Q1: 0.7758619613050001 Q2: 1.14470763225 Q3: 1.64722003606 Max: 10.3942654912

Conclusion: The distribution of column v120 is very highly positive/right skewed. Whereas in boxplot a very large number of outliers (both[q1-1.5*IQR] & [q3+1.5*IQR]) present in v120 with higher % at MAX points w .r.t both '0' and '1' case.

Col_name:v121

Pyplot representation of v121:



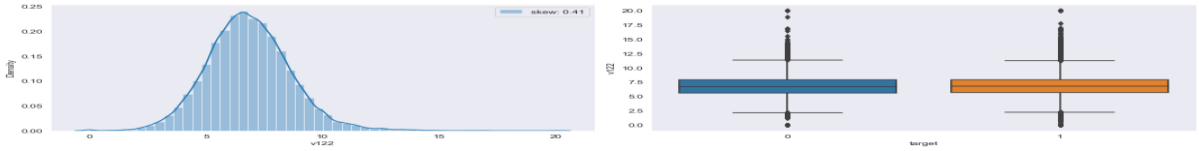
5-number statistics values of v121:

v121 Min: -9.82064247484e-07 Q1: 1.7869646964800001 Q2: 2.4361952032799996 Q3: 3.3791747336 Max: 20.00000080
35

Conclusion: The distribution of column v121 is highly positive/right skewed. Whereas in boxplot a very larg
e number of outliers (both[q1-1.5*IQR] & [q3+1.5*IQR]) present in v121 with higher % at MAX points w.r.t
both '0' and '1' case.

Col_name:v122

Pyplot representation of v122:



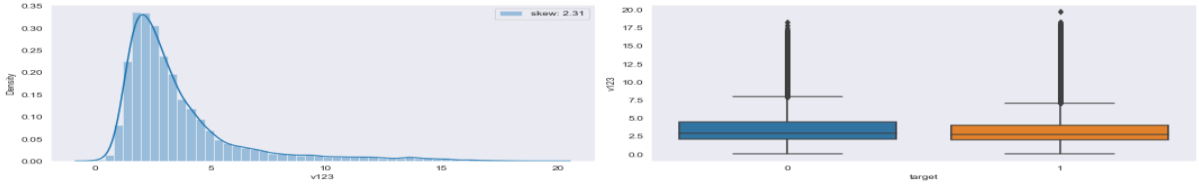
5-number statistics values of v122:

v122 Min: -9.97849689893e-07 Q1: 5.647711612004999 Q2: 6.749117430585 Q3: 7.911391804229999 Max: 20.00000093
24

Conclusion: The distribution of column v122 is very highly positive/right skewed. Whereas in boxplot a very
large number of outliers (both[q1-1.5*IQR] & [q3+1.5*IQR]) present in v122 with higher % at MAX points w
.r.t both '0' and '1' case.

Col_name:v123

Pyplot representation of v123:



5-number statistics values of v123:

v123 Min: 0.019138556878 Q1: 1.96331461253 Q2: 2.7392394849199997 Q3: 4.075361381455 Max: 19.6860692361

Conclusion: The distribution of column v123 is very much highly positive/right skewed. Whereas in boxplot
a very large number of outliers ([q3+1.5*IQR] region only) present in v123 with higher % at MAX points w.r
.t both '0' and '1' case.

Col_name:v124

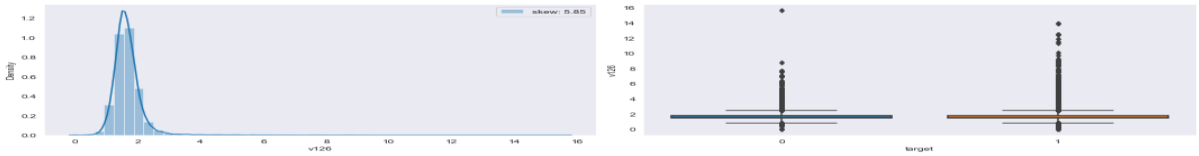
Pyplot representation of v124:

5-number statistics values of v124:

v124 Min: -9.994952909670001e-07 Q1: 0.0205377699201 Q2: 0.13986385687 Q3: 0.871833275401 Max: 20.0000009992
Conclusion: The distribution of column v124 is very much highly positive/right skewed. Whereas in boxplot a very large number of outliers (both[q1-1.5*IQR] & [q3+1.5*IQR]) present in v124 with higher % at MAX p oints w.r.t both '0' and '1' case.

Col_name:v126

Pyplot representation of v126:

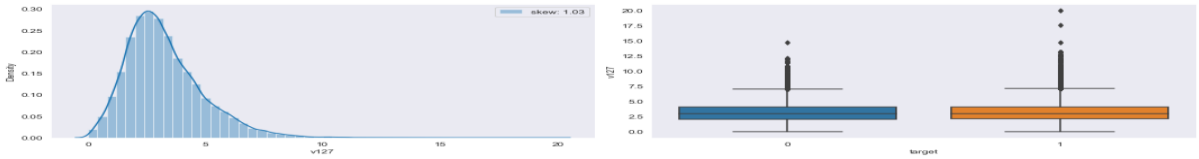


5-number statistics values of v126:

v126 Min: -9.56417383724e-07 Q1: 1.41759980088 Q2: 1.61480167446 Q3: 1.84388568748 Max: 15.6316128253
Conclusion: The distribution of column v126 is slightly positive/right skewed. Whereas in boxplot a very lar ge number of outliers (both[q1-1.5*IQR] & [q3+1.5*IQR]) present in v126 with higher % at MAX points w.r. t both '0' and '1' case.

Col_name:v127

Pyplot representation of v127:

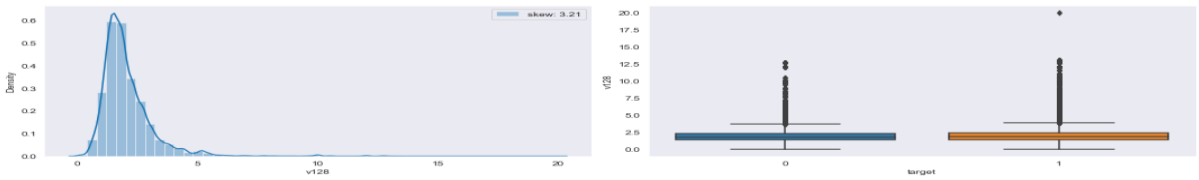


5-number statistics values of v127:

v127 Min: -9.22379751089e-07 Q1: 2.10190025732 Q2: 2.9636202722099996 Q3: 4.1081462427 Max: 19.9999990947
Conclusion: The distribution of column v127 is very much highly positive/right skewed. Whereas in boxplot a very large number of outliers ([q3+1.5*IQR] region only) present in v127 w.r.t both '0' and '1' case.

Col_name:v128

Pyplot representation of v128:



5-number statistics values of v128:

v128 Min: 8.19781164104e-07 Q1: 1.39382990954 Q2: 1.7984357936200002 Q3: 2.3901575351099997 Max: 20.000000402

Conclusion: The distribution of column v128 is very highly positive/right skewed. Whereas in boxplot a very large number of outliers ([q3+1.5*IQR]) present in v128 w.r.t both '0' and '1' case.

Col_name:v130

Pyplot representation of v130:

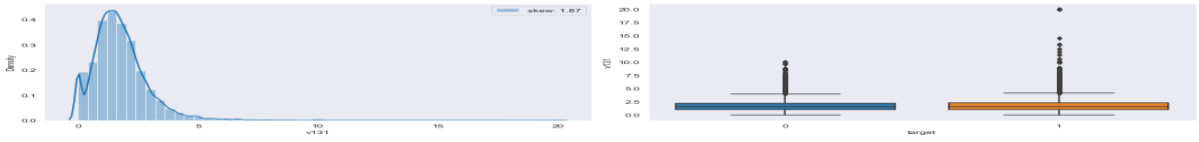
5-number statistics values of v130:

v130 Min: -9.90125732773e-07 Q1: 1.1061724012899998 Q2: 1.560137561955 Q3: 2.33242492849 Max: 19.9999995909

Conclusion: The distribution of column v130 is very much highly positive/right skewed. Whereas in boxplot a very large number of outliers ([q3+1.5*IQR] region only) present in v130 with higher % at MAX points w.r.t both '0' and '1' case.

Col_name:v131

Pyplot representation of v131:



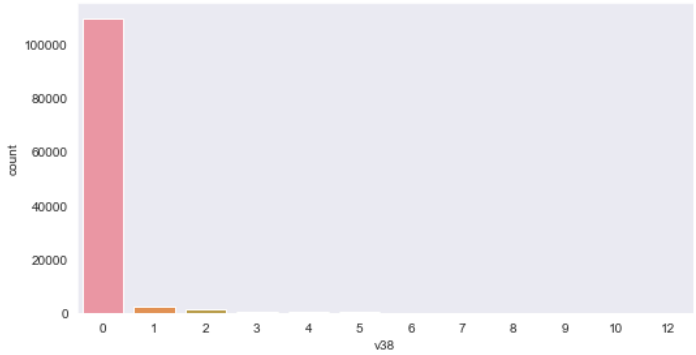
5-number statistics values of v131:

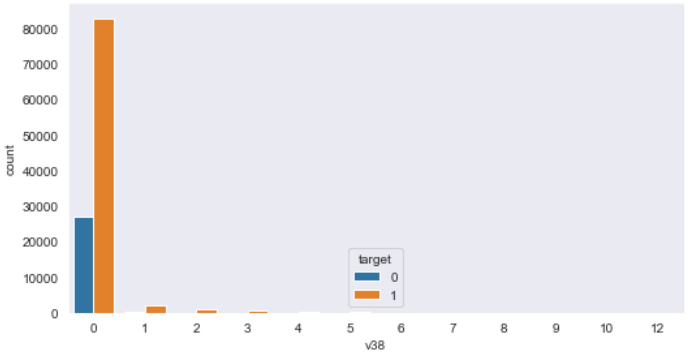
v131 Min: -9.99913392674e-07 Q1: 1.0126581477374998 Q2: 1.5894032727850003 Q3: 2.2619045881 Max: 20.000000942 6

Conclusion: The distribution of column v131 is highly positive/right skewed. Whereas in boxplot a very large number of outliers ([q3+1.5*IQR] region only) present in v131w.r.t both '0' and '1' case.

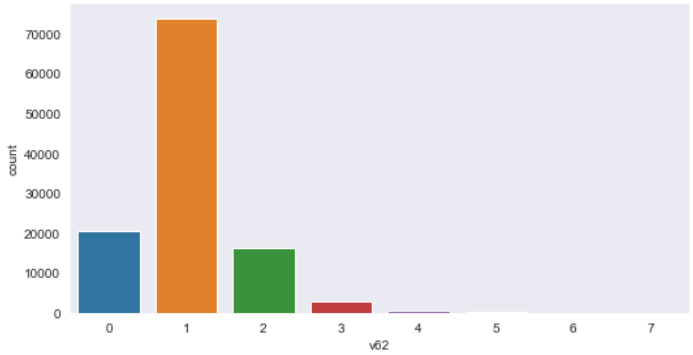
Integer Columns:

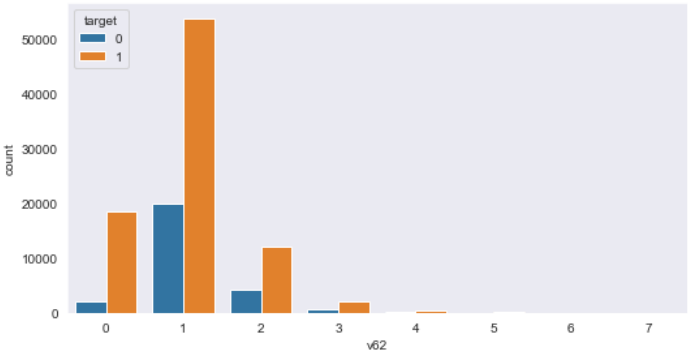
V38:



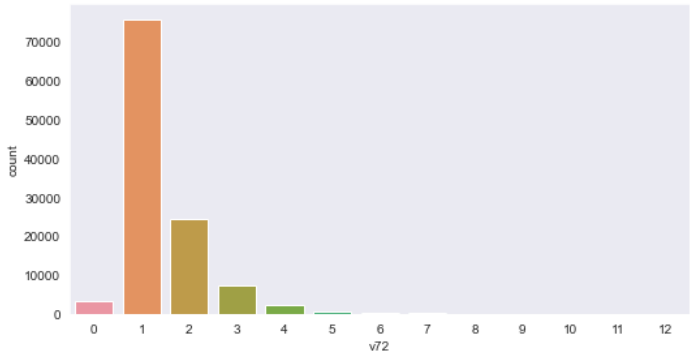


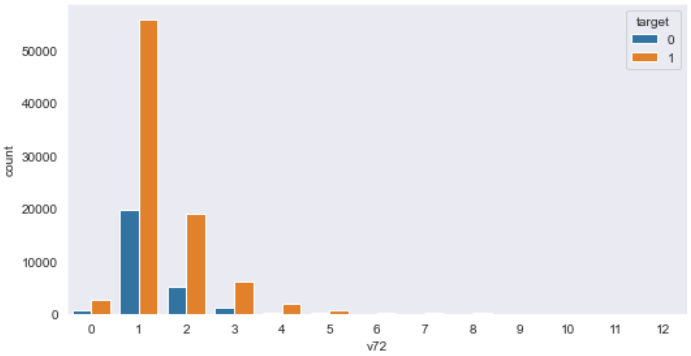
V62:



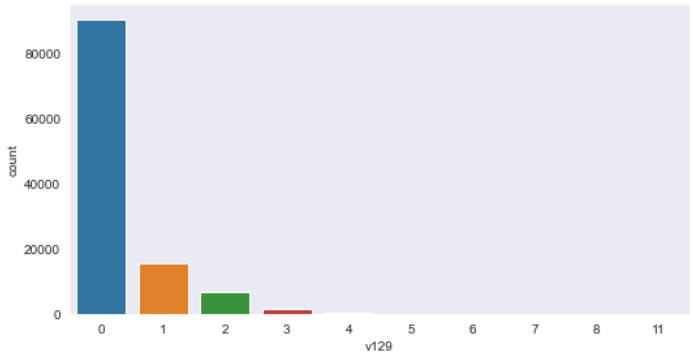


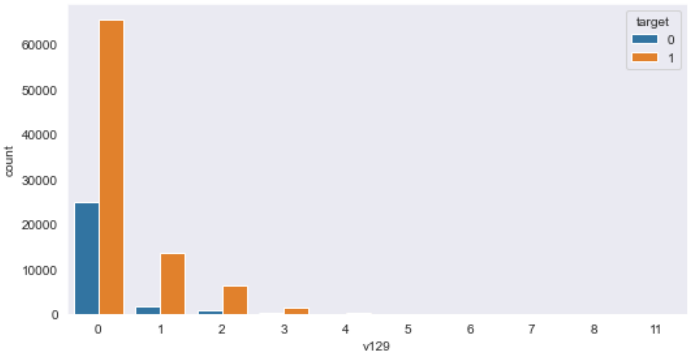
V72:





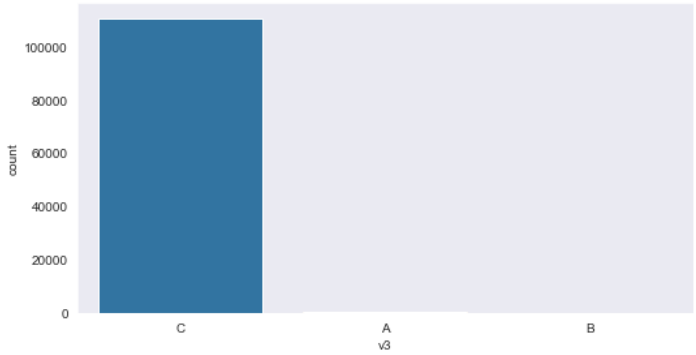
V129:

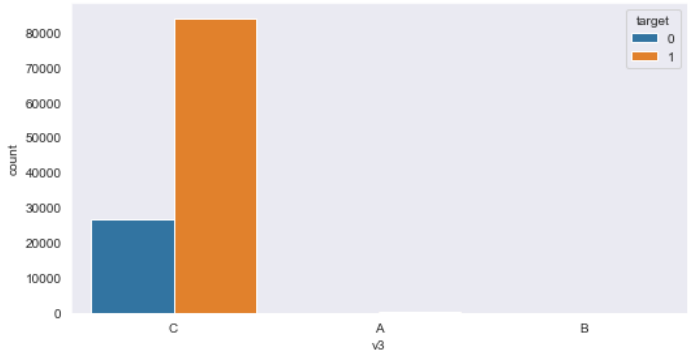




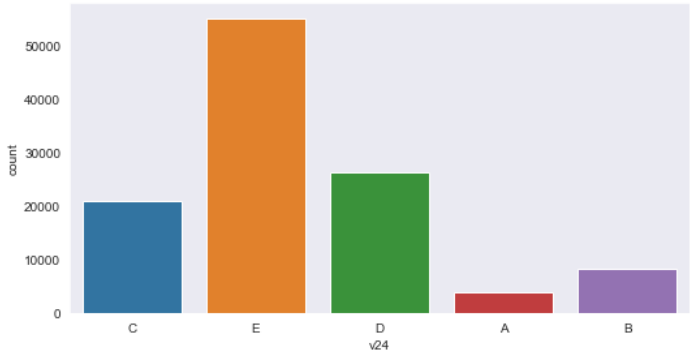
Object Columns:

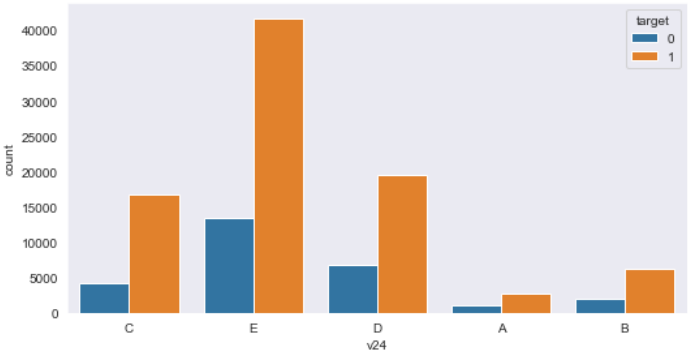
V3:



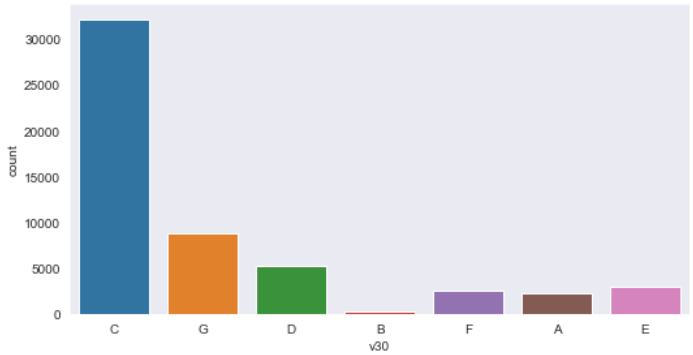


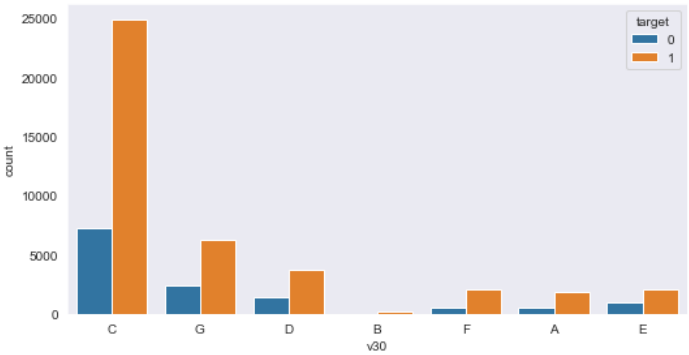
V24:



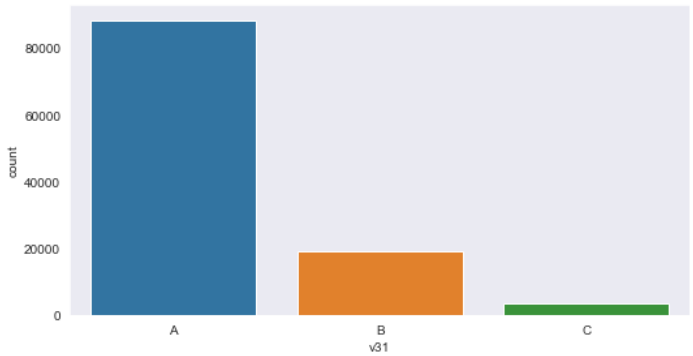


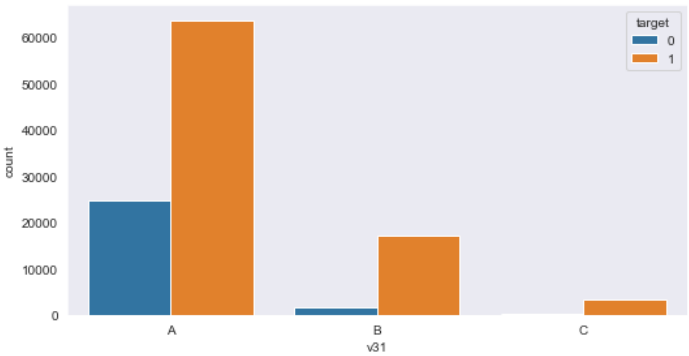
V30:



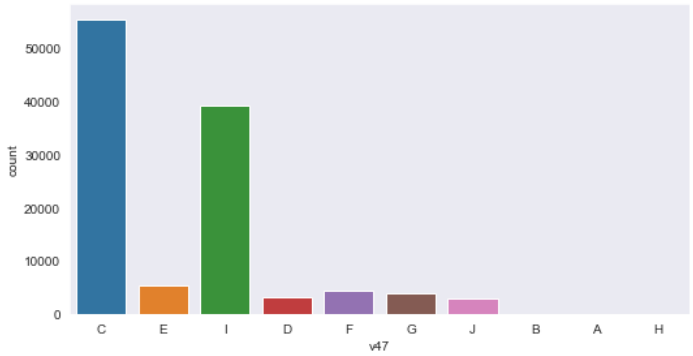


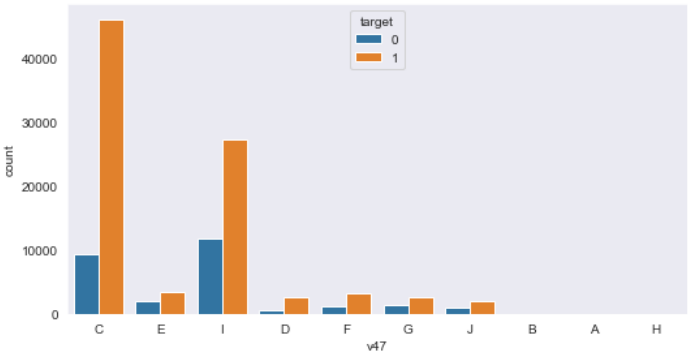
V31:



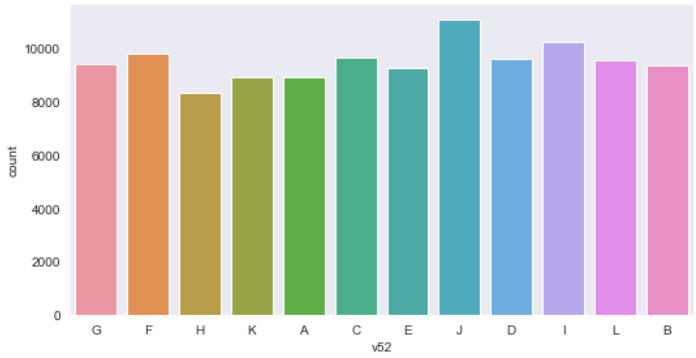


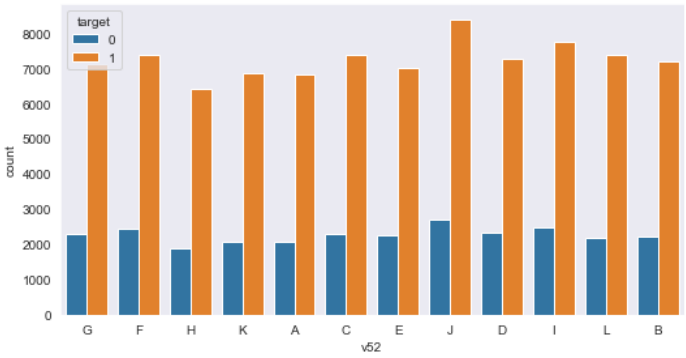
V47:



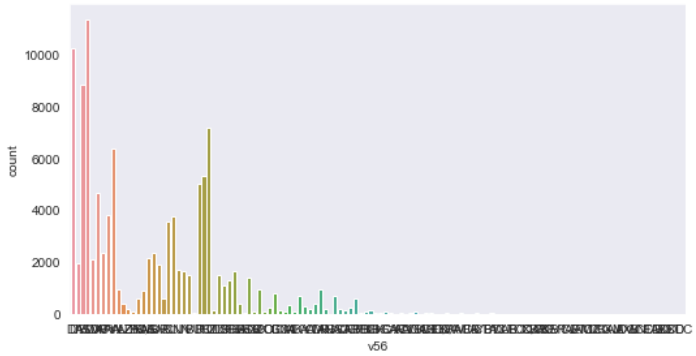


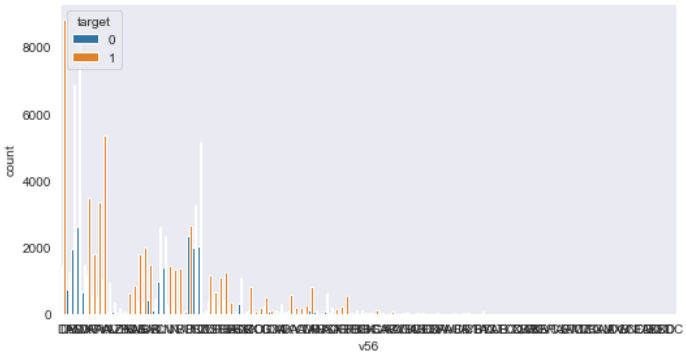
V52:



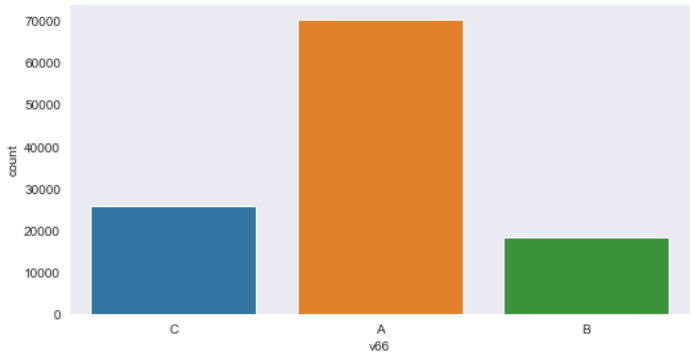


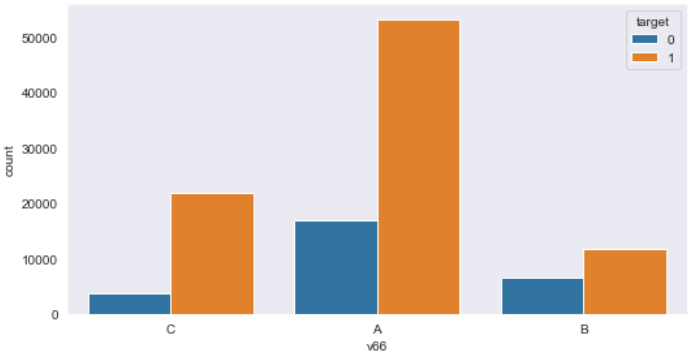
V56:



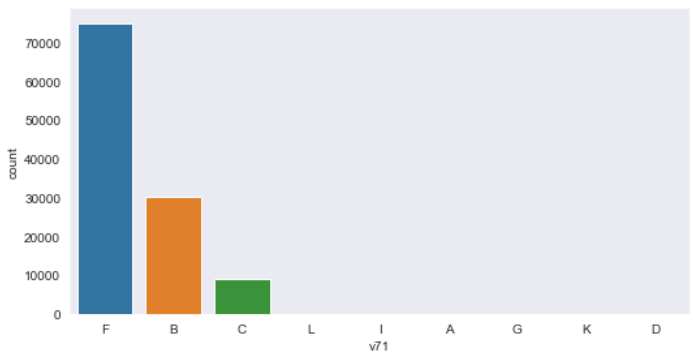


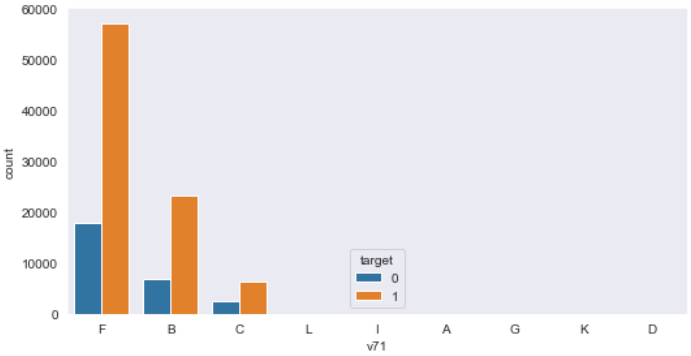
V66:



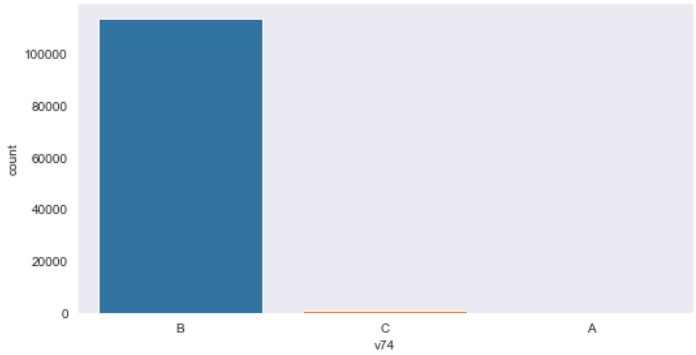


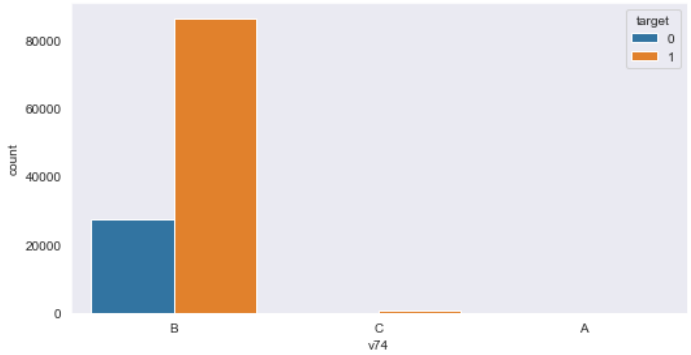
V71:



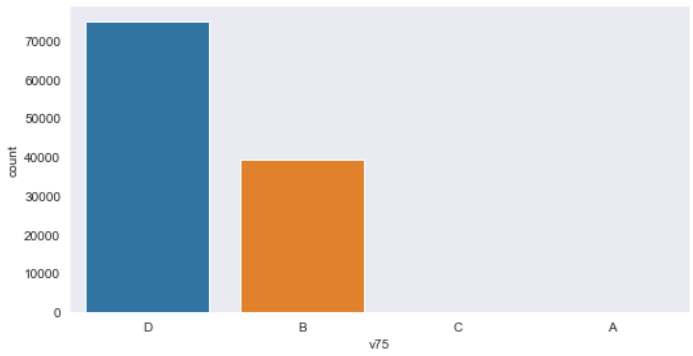


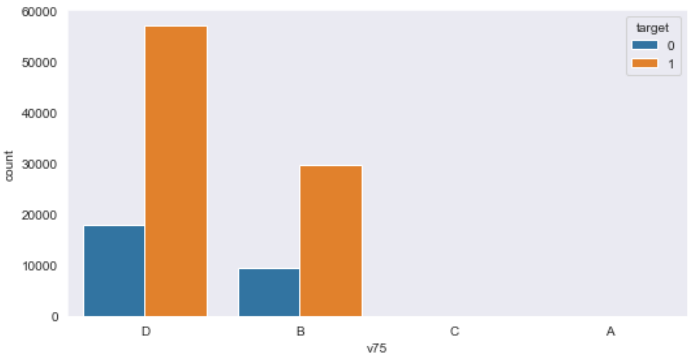
V74:



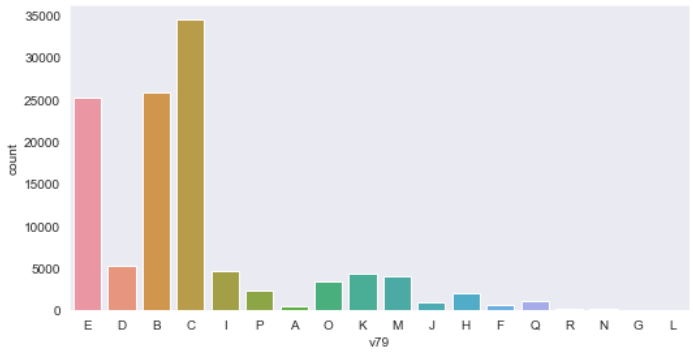


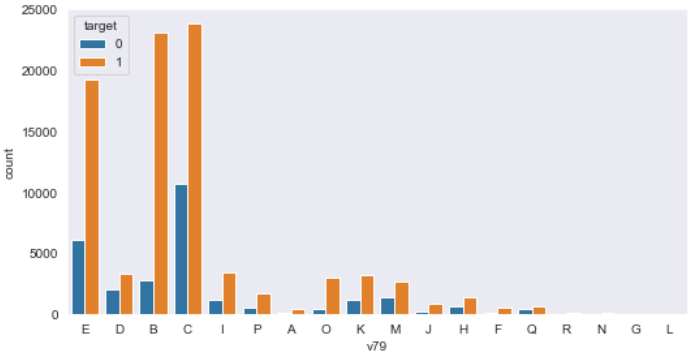
V75:



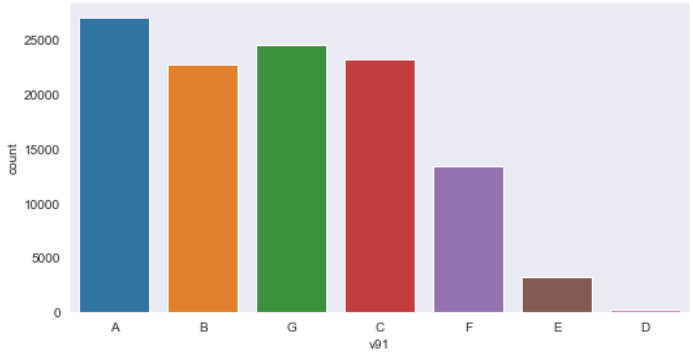


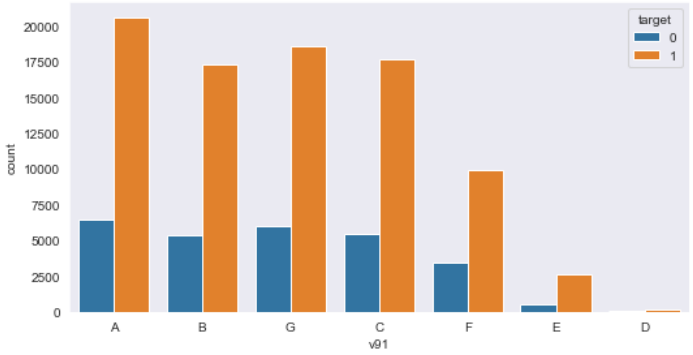
V79:



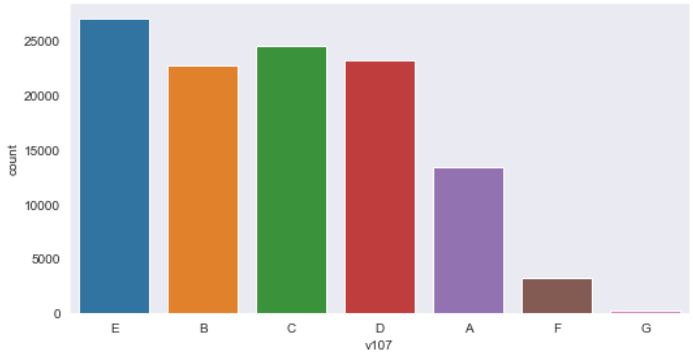


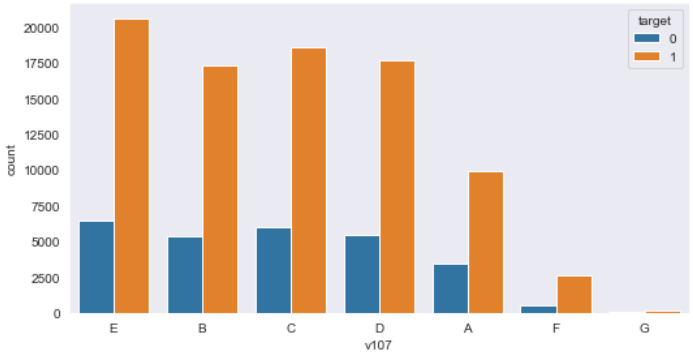
V91:



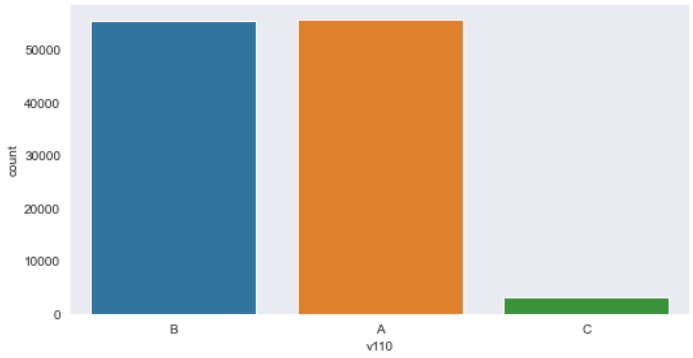


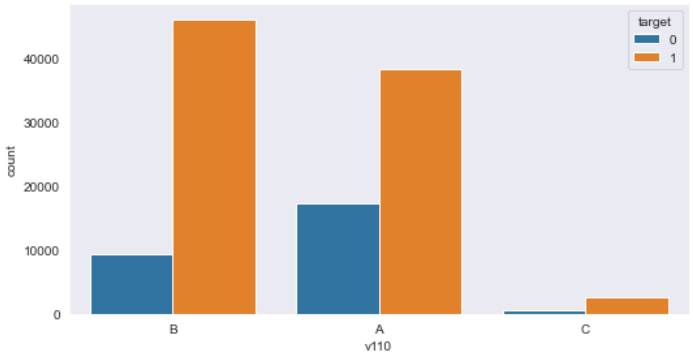
V107:



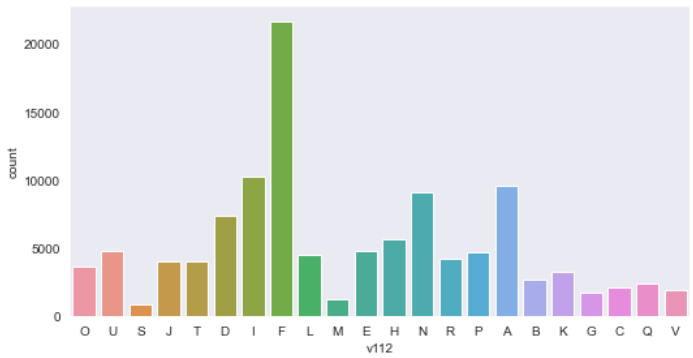


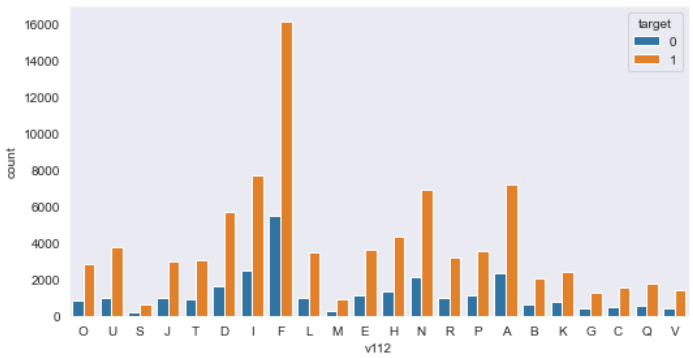
V110:



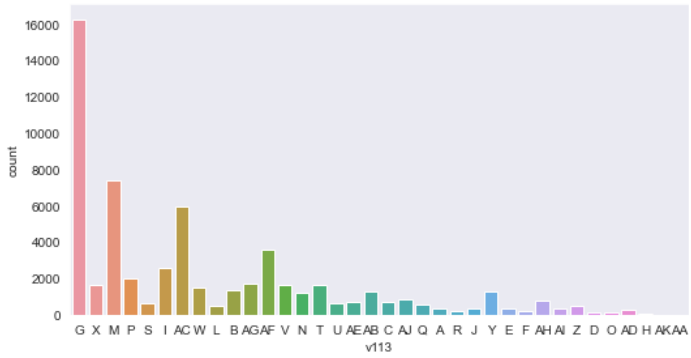


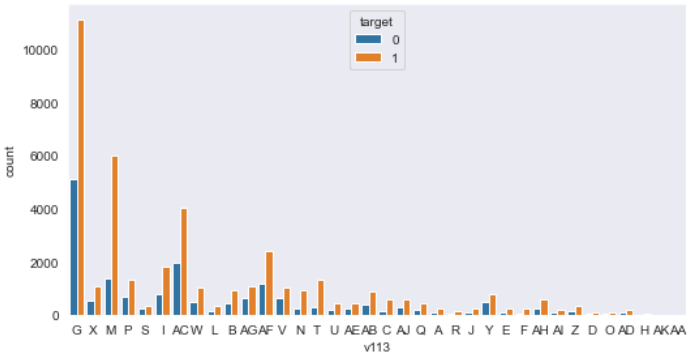
V112:



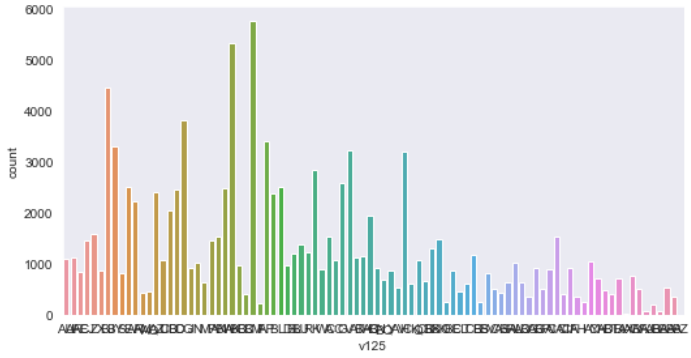


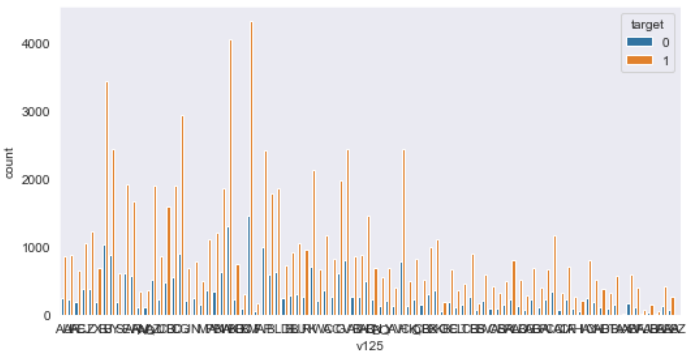
V113:


count
v113
G X M P S I AC W L B AG AF V N T U AE AB C AJ Q A R J Y E F AH AI Z D O AD H AK AA


target
0
1
count
v113
G X M P S I AC W L B AG AF V N T U AE AB C AJ Q A R J Y E F AH AI Z D O AD H AK AA

V125:


count
v125


target
0
1
count
v125

Data Cleaning

Column Drop Based on Crosstab:

Crosstab:

A cross-tabulation (or crosstab) is a two- (or more) dimensional table that records the number (frequency) of respondents that have the specific characteristics described in the cells of the table. Cross-tabulation tables provide a wealth of information about the relationship between the variables.

Shows the relationship between two categorical variables, a crosstab is also known as a contingency table.

From the EDA part we observe that 2 object column (“v3”,”v74”) and 1 int column (“v38”) is having one value count very high and other value count very low. So we print the normalized crosstab value of those column w.r.t. target column.

1. **Column v3:**

   - pd.crosstab(index=df["v3"],columns=df["target"],normalize="columns")

target	0	1
v3		
A	0.000376	0.002576
B	0.000225	0.000558
C	0.999399	0.996866

A and B value counts are negligible compare to C in v3 column.

So, we drop “v3” column from data set.

column "v3" is dropped

- df.drop("v3",axis=1,inplace=True)

```
print(df.shape)
```

```
(114321, 131)
```

2. **Column v74:**

- pd.crosstab(index=df["v74"],columns=df["target"],normalize="columns")

target	0	1
v74		
A	0.000256	0.000437
B	0.997473	0.992048
C	0.002271	0.007515

A and C value counts are negligible compare to B in v74 column.

So, we drop “v74” column from data set.

- df.drop("v74",axis=1,inplace=True)
  print(df.shape)

(114321, 130)

3. **Column v38:**

- pd.crosstab(index=df["v38"],columns=df["target"],normalize="columns")

target	0	1
v38		
0	0.987106	0.951219
1	0.006923	0.022454
2	0.003297	0.012733
3	0.001062	0.005803
4	0.000513	0.002781
5	0.000476	0.001609
6	0.000073	0.000770
7	0.000183	0.000919

8	0.000000	0.000701
9	0.000037	0.000425
10	0.000330	0.000437
12	0.000000	0.000149

Other value counts are negligible compare to 0 in v38 column.

So, we drop “v38” column from data set.

- df.drop("v38",axis=1,inplace=True)

  print(df.shape)

  (114321, 129)

As the numbers of columns are very high and conlumn names are masked for the confidentiality of the customer details, we can’t select appropriate feature by the conventional EDA. We have to go for model-based feature selection.

Filling Null values and finding WOE values:

WOE:

The weight of evidence tells the predictive power of an independent variable in relation to the dependent variable. Since it evolved from credit scoring world, it is generally described as a measure of the separation of good and bad customers. "Bad Customers" refers to the customers who defaulted on a loan. and "Good Customers" refers to the customers who paid back loan.

$$WOE = \ln\left(\frac{\text{Distribution of Goods}}{\text{Distribution of Bads}}\right)$$

WOE Calculation

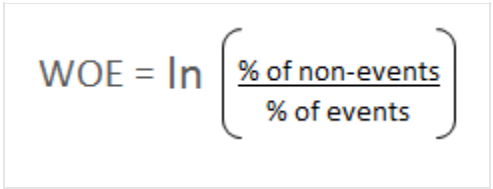
Distribution of Goods - % of Good Customers in a particular group
Distribution of Bads - % of Bad Customers in a particular group
ln - Natural Log

Positive WOE means Distribution of Goods > Distribution of Bads

Negative WOE means Distribution of Goods < Distribution of Bads

Hint : Log of a number > 1 means positive value. If less than 1, it means negative value.
. It's good to understand the concept of WOE in terms of **events and non-events**. It is calculated by taking the natural logarithm (log to base e) of division of % of non-events and % of events.

WOE = In(% of non-events ÷ % of events)



Weight of Evidence Formula

Steps of Calculating WOE

1.For a continuous variable, split data into 10 parts (or lesser depending on the distribution).

2.Calculate the number of events and non-events in each group (bin)

3.Calculate the % of events and % of non-events in each group.

4.Calculate WOE by taking natural log of division of % of non-events and % of events

Note : For a categorical variable, we have no need to split the data.

Terminologies related to WOE

1. Fine Classing

Creation of 10/20 bins/groups for a continuous independent variable and then calculates WOE and IV of the variable

2. Coarse Classing

Combination of adjacent categories with similar WOE scores

Usage of WOE

Weight of Evidence (WOE) helps to transform a continuous independent variable into a set of groups or bins based on similarity of dependent variable distribution i.e. number of events and non-events.

1.Float columns:

After taking several approaches to fill the null values of float columns, finally we found that filling null values with mean value is best suited in our project.

And while filling null values with mean we fill the null values corresponding to target value=0 with the mean of the values corresponding to target value=0 and the null values corresponding to target value=1 with the mean of the values corresponding to target value=1.

Code for so is→

```
for f in df.columns:
   if df[f].dtype == 'float64':
      m = df.groupby("target")[f].mean()
```

```
df.loc[df.target==0,f]=df.loc[df.target==0,f].fillna(m[0])  # target=0 rows null value replaced with mean of all target=0 non null rows

df.loc[df.target==1,f]=df.loc[df.target==1,f].fillna(m[1])  # target=0 rows null value replaced with mean of all target=0 non null rows
```

As most of the columns having null value was float columns only. So, now let's see what are the columns left with null value and their null value percentage.

Code→

```
nulcols = []
print("Columns with null value(percentage):")
for col in df:
   if(df[col].isna().sum())!=0:
      print(col,(((df[col].isna().sum())/df.shape[0])*100))
      nulcols.append(col)
```

```
Columns with null value(percentage):
v22 0.43736496356749854
v30 52.580015920084676
v31 3.023941358105685
v52 0.0026241897814049914
v56 6.01989135854305
v91 0.0026241897814049914
v107 0.0026241897814049914
v112 0.3341468321655689
v113 48.376063890273876
v125 0.06735420438939478
```

```
df[nulcols][:5]
```

	v22	v30	v31	v52	v56	v91	v107	v112	v113	v125
0	XDX	C	A	G	DI	A	E	O	NaN	AU
1	GUV	C	A	G	DY	B	B	U	G	AF
2	FQ	NaN	A	F	AS	G	C	S	NaN	AE
3	ACUE	C	B	H	BW	B	B	J	NaN	CJ
4	HIT	NaN	A	H	NaN	G	C	T	G	Z
5	AYX	NaN	A	K	DX	G	C	D	X	X

So, we can see that all the columns having null value is of type object.

2.Object Columns:

We are planning to replace all the object column and int column values with “Weight of Evidence” (woe) values.

So, for that we develop a function which takes data frame and column name as input and replaces the column values to woe values in the data frame itself.

Function code→

```
def woefun(df,col):
    woedf=pd.crosstab(index=df[col],columns=df["target"])
    woedf.rename(columns={0:"zero",1:"one"},inplace=True)
    s1 = woedf["zero"].sum()
    s2 = woedf["one"].sum()
    for i in woedf.index:
        if woedf["zero"][i]==0 or woedf["one"][i]==0:
            woedf["zero"][i]=woedf["zero"][i]+1
            woedf["one"][i]=woedf["one"][i]+1
    woedf["woe"]=np.log(woedf["zero"]/s1)-np.log(woedf["one"]/s2)
    d = dict(woedf["woe"])
    df[col].replace(d,inplace=True)
```

Now, we will use this function to replace column values to woe in two step.

1. **Object Columns with null value:**

We have those column names in “nullcols” list.

We fill the Null values with “missing” string for woe calculation.

```
for col in nulcols:
    df[col][df[col] != df[col]] = "missing"
```

And now we call the function to replace all values with woe value.

```
for col in nulcols:
    woefun(df,col)
```

```
df[nulcols][:5]
```

	v22	v30	v31	v52	v56	v91	v107	v112	v113	v125
0	0.466116	-0.074482		0.210184	0.034001	-0.667281		-0.002338		-0.002338
	-0.075504		-0.383614		-0.076209					
1	-0.023433		-0.074482	0.210184	0.034001	0.598052	-0.012831			-0.012831
	-0.150208		0.380882	-0.173024						
2	0.060650	-0.011506		0.210184	0.049470	-0.098530		0.023750	0.023750	-0.046271
	-0.383614		-0.046271							
3	-0.307074		-0.074482	-1.167479		-0.065601		-0.054893		-0.012831
	-0.012831		0.070374	-0.383614		0.149076				
4	-1.238633		-0.011506	0.210184	-0.065601		0.659118	0.023750	0.023750	-0.042534
	0.380882	-0.030906								
5	0.466116	-0.011506		0.210184	-0.029135		0.331212	0.023750	0.023750	-0.088156
	0.490786	-0.166149								

2. **Object column without null value:**

Let's see the object column left in the data set and replace those values with woe value.

```
objcols_list = []
objcols=df.select_dtypes(include="object").columns
for col in objcols:
    objcols_list.append(col)

print(objcols_list)
['v24', 'v47', 'v66', 'v71', 'v75', 'v79', 'v110']

for col in objcols_list:
    woefun(df,col)
df[objcols_list][:5]
```

	v24	v47	v66	v71	v75	v79	v110

0 -0.224458 -0.437047 -0.609493 -0.007233 -0.007478
0.012721-0.437069

1 -0.224458 0.6380610.017872-0.007233 -0.007478 0.6377600.362727

2 0.026307-0.437047 0.017872-0.066987 0.0136930.012721-0.437069

3 0.090355-0.437047 0.017872-0.007233 -0.007478 -0.957114 -
0.437069

4 0.0263070.329055-0.609493 -0.007233 -0.007478 0.3594260.362727

5 0.1538610.3290550.017872-0.007233 -0.007478 0.0731820.362727

So all the object columns are converted with woe value and type is converted to float64.

3.Integer Columns:

We are left with only 3 int columns. ("v62","v72","v129")

So lets call woe function to convert all int values with woe value.

```
intcols_list = ["v62","v72","v129"]
for col in intcols_list:
    woefun(df,col)
df[intcols_list][:5]
```

v62 v72 v129

0 0.1704670.1206610.185527

1 0.137418-0.130512 0.185527

2 0.170467-0.501055 -0.998118

3 0.170467-0.130512 -0.927768

4 0.1704670.1206610.185527

5 0.1704670.1206610.185527

Train Test Split:

Now our data set is left with 129 columns and all the columns are converted to float type. So, now we are dividing our column to X and y for further progress.

```
# differentiating X and y for models

y = df["target"]

X = df.drop("target",axis=1)

# spliting dataset into train and test data

Xtrain,Xtest,ytrain,ytest=model_selection.train_test_split(X,y,test_size=.2,random_state=100)

Xtrain.shape
```

(91456, 128)

```
Xtest.shape
```

(22865, 128)

Data Pre-processing And Feature Selection

Data preprocessing is a process of preparing the raw data and making it suitable for a machine learning model. It is the first and crucial step while creating a machine learning model.

When creating a machine learning project, it is not always a case that we come across the clean and formatted data. And while doing any operation with data, it is mandatory to clean it and put in a formatted way. So for this, we use data preprocessing task.

Importance:

A real-world data generally contains noises, missing values, and maybe in an unusable format which cannot be directly used for machine learning models. Data preprocessing is required tasks for cleaning the data and making it suitable for a machine learning model which also increases the accuracy and efficiency of a machine learning model.

It involves below steps:

- Getting the dataset
- Importing libraries
- Importing datasets
- Finding Missing Data
- Encoding Categorical Data
- Splitting dataset into training and test set
- Feature scaling

We already saved all the floating columns of original dataset in floatcols. And now we are pre-processing floating columns to get reduce all the values and to get better result. For that we import required library and code.

Code→

```
# importing library for pre-processing of dataa and creating object of that
from sklearn import preprocessing
scaler=preprocessing.StandardScaler()
Xtrain[floatcols]=scaler.fit_transform(Xtrain[floatcols])  # fit-transform data to get normalized values
Xtest[floatcols]=scaler.fit_transform(Xtest[floatcols])
```

Base Model Building:

Now out datasets are ready for testing with a base model and for further work. Base Model is very important because based on this model we will go for further feature reduction and try to improve scores of our model. So we 1st build a base model of "Logistic Regression" to get an idea on various scores with all 128 columns.

Code→

```
model=linear_model.LogisticRegression()    # object of logistic regression model
model.fit(Xtrain,ytrain)    # fitting model
trainp=model.predict(Xtrain)
testp=model.predict(Xtest)
from sklearn import metrics  # to show the scores of model build
print("Accuracy of training",metrics.accuracy_score(ytrain,trainp))
print("Accuracy of test",metrics.accuracy_score(ytest,testp))
print("Precision of training",metrics.precision_score(ytrain,trainp))
print("Precision of test",metrics.precision_score(ytest,testp))
print("Confusion matrix of training",metrics.confusion_matrix(ytrain,trainp))
print("Confusion matrix of test",metrics.confusion_matrix(ytest,testp))
print("AUC of training",metrics.roc_auc_score(ytrain,trainp))
print("AUC of test",metrics.roc_auc_score(ytest,testp))
print("F1 score of training",metrics.f1_score(ytrain,trainp))
print("F1 score of test",metrics.f1_score(ytest,testp))
```

```
Accuracy of training 0.8042118614415675
Accuracy of test 0.7991690356439973
Precision of training 0.8196118135124457
Precision of test 0.8156031879609921
Confusion matrix of training [[ 7168 14610]
 [ 3296 66382]]
Confusion matrix of test [[ 1797  3725]
 [  867 16476]]
AUC of training 0.6409180945335436
AUC of test 0.6377171097340757
F1 score of training 0.8811574965155639
F1 score of test 0.877690176859152
```

Now we get a basic idea about our Machine learning model prediction.

Recursive Feature Elemination(RFE):

Recursive feature elimination (RFE) is a feature selection method that fits a model and removes the weakest feature (or features) until the specified number of features is reached. Features are ranked by the model's coef or feature_importances attributes, and by recursively eliminating a small number of features per loop, RFE attempts to eliminate dependencies and collinearity that may exist in the model.

RFE requires a specified number of features to keep, however it is often not known in advance how many features are valid. To find the optimal number of features cross-validation is used with RFE to score different feature subsets and select the best scoring collection of features.

Now we decided to go for further with almost 50% columns. For that we call RFE and select 60 columns from 129 columns.

Code→

```
# Recursive feature elimination to reduce features to 60
from sklearn import feature_selection   # importing required library
logreg=linear_model.LogisticRegression()
rfeobj=feature_selection.RFE(logreg,60)    # creating object to fit model
rfeobj.fit(Xtrain,ytrain)    # fitting model to get reduced feature
cols=Xtrain.columns[rfeobj.support_]    # taking the 60 selected columns in cols
cols
```

```
Index(['v7', 'v8', 'v9', 'v10', 'v11', 'v12', 'v15', 'v16', 'v19', 'v21',
    'v22', 'v24', 'v25', 'v26', 'v28', 'v30', 'v31', 'v36', 'v40', 'v41',
    'v43', 'v46', 'v47', 'v49', 'v50', 'v52', 'v53', 'v56', 'v58', 'v60',
    'v62', 'v65', 'v66', 'v70', 'v71', 'v72', 'v73', 'v75', 'v77', 'v78',
    'v79', 'v80', 'v84', 'v88', 'v89', 'v91', 'v97', 'v98', 'v100', 'v105',
    'v110', 'v111', 'v112', 'v113', 'v116', 'v117', 'v121', 'v123', 'v125',
    'v129'],
   dtype='object')
```

Now again we create "Logistic Regression" model to see the scores with left over 60 columns.

Code→

```
# again we are testing the result with above 60 columns
model=linear_model.LogisticRegression()
model.fit(Xtrain[cols],ytrain)
trainp=model.predict(Xtrain[cols])
testp=model.predict(Xtest[cols])
print("Accuracy of training",metrics.accuracy_score(ytrain,trainp))
print("Accuracy of test",metrics.accuracy_score(ytest,testp))
print("Precision of training",metrics.precision_score(ytrain,trainp))
print("Precision of test",metrics.precision_score(ytest,testp))
print("Confusion matrix of training",metrics.confusion_matrix(ytrain,trainp))
print("Confusion matrix of test",metrics.confusion_matrix(ytest,testp))
print("AUC of training",metrics.roc_auc_score(ytrain,trainp))
print("AUC of test",metrics.roc_auc_score(ytest,testp))
print("F1 score of training",metrics.f1_score(ytrain,trainp))
print("F1 score of test",metrics.f1_score(ytest,testp))
# the scores remain almost same
```

```
Accuracy of training 0.8039494401679497
Accuracy of test 0.7989503608134704
Precision of training 0.819321715950042
Precision of test 0.8157451446690448
Confusion matrix of training [[ 7138 14640]
 [ 3290 66388]]
Confusion matrix of test [[ 1803  3719]
 [  878 16465]]
AUC of training 0.6402723812483516
AUC of test 0.6379432603229682
F1 score of training 0.8810266346396295
F1 score of test 0.8775015322301277
```

As we can see that the scores remains almost same. So, we are now ready to go for checking of corelations among columns and remove them to get a better model.

Variance Inflation Factor(VIF):

The Variance Inflation Factor (VIF) is a measure of collinearity among predictor variables within a multiple regression.

The **variance inflation factor** (**VIF**) quantifies the extent of correlation between one predictor and the other predictors in a model. It is used for diagnosing collinearity/multicollinearity. Higher values signify that it is difficult to impossible to assess accurately the contribution of predictors to a model.

It is calculated by taking the ratio of the variance of all a given model's betas divide by the variance of a single beta if it were fit alone.

Collinearity is the state where two variables are highly correlated and contain similar information about the variance within a given dataset. To detect collinearity among variables, simply create a correlation matrix and find variables with large absolute values.

Now we write code to find out vif and p>|z| values and we observe them to select the useless columns and again we remove them recursively to get all the vif values<5 and p>|z| values <0.05.

1. First we do same for 60 columns:

Code→

```
# finding out "Variance Inflation Factor" to further discard some columns

import statsmodels.api as sm

from statsmodels.stats.outliers_influence import variance_inflation_factor

vif = pd.DataFrame()  # blank data frame

vif['Features'] = Xtrain[cols].columns

vif['VIF'] = [variance_inflation_factor(Xtrain[cols].values, i) for i in range(Xtrain[cols].shape[1])]

vif['VIF'] = round(vif['VIF'], 2)

vif = vif.sort_values(by = "VIF", ascending = False)

Xtrainsm=sm.add_constant(Xtrain[cols])

logm1 = sm.GLM(ytrain,Xtrainsm, family = sm.families.Binomial())    #
model=linear_model.LogisticRegression()

logm1results=logm1.fit()    # fit() return an result object

logm1results.summary()
```

Now we select 23 columns as useless columns.
["v11","v15","v25","v41","v43","v47","v49","v53","v58","v60","v71","v73","v77","v80","v84","v88","v89","v91","v97","v100","v111","v116","v121"]

And again we run “Logistic Regression” model with left over 37 columns to see the scores.

Code→

```
useless_cols =
["v11","v15","v25","v41","v43","v47","v49","v53","v58","v60","v71","v73","v77","v80","v84","v88",
"v89","v91","v97","v100","v111","v116","v121"]

lst = []

for i in cols:

  if i in useless_cols:

    continue

  lst.append(i)

cols = lst

print(len(cols))
```

Now after model building the scores are:

```
Accuracy of training 0.8041134534639608
Accuracy of test 0.7980319265252569
Precision of training 0.8191582504038573
Precision of test 0.8144304423029405
Confusion matrix of training [[ 7113 14665]
 [ 3250 66428]]
Confusion matrix of test [[ 1767  3755]
 [  863 16480]]
AUC of training 0.6399854421584714
AUC of test 0.6351160229373376
F1 score of training 0.8811774147548267
F1 score of test 0.8771089467241471
```

They remain again almost same.

2. <u>Now again we Do the same vif work with 37 columns:</u>

Now we select 7 columns as useless columns. ["v7","v9","v26","v46","v65","v98","v105"]

And again we run “Logistic Regression” model with left over 30 columns to see the scores.

Code→

```
useless_cols = ["v7","v9","v26","v46","v65","v98","v105"]

lst = []
```

```
for i in cols:
    if i in useless_cols:
        continue
    lst.append(i)
cols = lst
print(len(cols))
```

Now after model building the scores are:

```
Accuracy of training 0.804015045486354
Accuracy of test 0.7984255412202056
Precision of training 0.8190909539311433
Precision of test 0.8146061863820535
Confusion matrix of training [[ 7107 14671]
 [ 3253 66425]]
Confusion matrix of test [[ 1770  3752]
 [  857 16486]]
AUC of training 0.6398261608636995
AUC of test 0.6355606441004341
F1 score of training 0.8811200870176555
F1 score of test 0.8773582395359357
```

They remain again almost same again.

3. Following the same process again we remove 4 another columns:

["v8",”v10”,"v31","v110"]

Now we are left with 26 columns.[v12 v16 v19 v21 v22 v24 v28 v30 v36 v40 v50 v52 v56 v62 v66 v70 v72 v75 v78 v79 v112 v113 v117 v123 v125 v129

]

And again we run “Logistic Regression” model with left over 26 columns to see the scores.

Now after model building the scores are:

```
Accuracy of training 0.804277466759972
Accuracy of test 0.7985130111524164
Precision of training 0.8191838244359512
Precision of test 0.8146555983792865
Confusion matrix of training [[ 7112 14666]
 [ 3234 66444]]
Confusion matrix of test [[ 1771  3751]
 [  856 16487]]
```

```
AUC of training 0.6400772970671056
AUC of test 0.6356800210792648
F1 score of training 0.8812902883518583
F1 score of test 0.8774114579175647
```

They remain again almost same.

And all out vif and p>|z| values are less than 5 and 0.05 respectively.

	Features	VIF
1	v16	3.11
18	v78	3.03
19	v79	3.02
25	v129	2.29
13	v62	2.15
21	v113	1.81
20	v112	1.77
24	v125	1.71
12	v56	1.71
0	v12	1.58
8	v36	1.56
14	v66	1.51
9	v40	1.48
22	v117	1.40
23	v123	1.37
15	v70	1.33
10	v50	1.32

Features	VIF	
16	v72	1.20
3	v21	1.20
6	v28	1.13
2	v19	1.09
5	v24	1.05
4	v22	1.04
7	v30	1.04
17	v75	1.01
11	v52	1.00

Dep. Variable:	target	**No. Observations:**	91456
Model:	GLM	**Df Residuals:**	91429
Model Family:	Binomial	**Df Model:**	26
Link Function:	logit	**Scale:**	1.0000
Method:	IRLS	**Log-Likelihood:**	-39868.
Date:	Thu, 08 Jul 2021	**Deviance:**	79737.
Time:	18:51:51	**Pearson chi2:**	9.88e+04
No. Iterations:	6		
Covariance Type:	nonrobust		

	coef	std err	z	P>|z|	[0.025	0.975]
const	1.5929	0.011	144.534	0.000	1.571	1.615
v12	-0.0374	0.013	-2.891	0.004	-0.063	-0.012
v16	-0.0726	0.016	-4.437	0.000	-0.105	-0.041
v19	0.0611	0.018	3.454	0.001	0.026	0.096
v21	0.0753	0.010	7.230	0.000	0.055	0.096
v22	-1.3399	0.016	-85.287	0.000	-1.371	-1.309
v24	-0.6116	0.085	-7.212	0.000	-0.778	-0.445
v28	-0.0908	0.009	-9.673	0.000	-0.109	-0.072
v30	-0.2294	0.075	-3.051	0.002	-0.377	-0.082
v36	-0.0930	0.011	-8.270	0.000	-0.115	-0.071

v40	0.0614	0.011	5.783	0.000	0.041	0.082
v50	0.7092	0.014	50.179	0.000	0.682	0.737
v52	-0.8616	0.238	-3.620	0.000	-1.328	-0.395
v56	-0.3491	0.020	-17.365	0.000	-0.388	-0.310
v62	0.1242	0.036	3.432	0.001	0.053	0.195
v66	-0.7481	0.030	-25.165	0.000	-0.806	-0.690
v70	0.0211	0.010	2.067	0.039	0.001	0.041
v72	0.1519	0.046	3.301	0.001	0.062	0.242
v75	-2.0040	0.635	-3.158	0.002	-3.248	-0.760
v78	-0.0741	0.016	-4.574	0.000	-0.106	-0.042
v79	-0.2952	0.029	-10.078	0.000	-0.353	-0.238
v112	-0.8453	0.195	-4.342	0.000	-1.227	-0.464
v113	-0.1983	0.029	-6.930	0.000	-0.254	-0.142
v117	-0.0890	0.011	-8.365	0.000	-0.110	-0.068
v123	-0.0944	0.010	-9.796	0.000	-0.113	-0.075
v125	0.3589	0.119	3.017	0.003	0.126	0.592
v129	-0.2578	0.033	-7.853	0.000	-0.322	-0.193

So, we decided to go for final model building with these 26 columns.

Model Building

Now there are two Types of ML Classification Algorithms:

1. Linear Models- Logistic Regression. Support Vector Machines.

2. Non-linear Models- K-Nearest Neighbours. Kernel SVM. Naïve Bayes. Decision Tree Classification. Random Forest Classification.

Evaluating a Classification model:

Once our model is completed, it is necessary to evaluate its performance; either it is a Classification or Regression model. So, for evaluating a Classification model, we have the following ways:

1. Log Loss or Cross-Entropy Loss:

- It is used for evaluating the performance of a classifier, whose output is a probability value between the 0 and 1.
- For a good binary Classification model, the value of log loss should be near to 0.
- The value of log loss increases if the predicted value deviates from the actual value.
- The lower log loss represents the higher accuracy of the model.
- For Binary classification, cross-entropy can be calculated as:

1. ?(ylog(p)+(1?y)log(1?p))

Where y= Actual output, p= predicted output.

2. Confusion Matrix:

- The confusion matrix provides us a matrix/table as output and describes the performance of the model.
- It is also known as the error matrix.
- The matrix consists of predictions result in a summarized form, which has a total number of correct predictions and incorrect predictions. The matrix looks like as below table:

○	**Actual Positive**	**Actual Negative**
Predicted Positive	True Positive	False Positive

Predicted Negative	False Negative	True Negative

$$\text{Accuracy} = \frac{\text{TP+TN}}{\text{Total Population}}$$

3. AUC-ROC curve:

- ROC curve stands for **Receiver Operating Characteristics Curve** and AUC stands for **Area Under the Curve**.
- It is a graph that shows the performance of the classification model at different thresholds.
- To visualize the performance of the multi-class classification model, we use the AUC-ROC Curve.
- The ROC curve is plotted with TPR and FPR, where TPR (True Positive Rate) on Y-axis and FPR(False Positive Rate) on X-axis.

For an insurance company a person not eligible for claim(0), if he/she is approved for claim(1), this scenario is not intended. So we have to reduce the number of "False +ve" cases.

So "Precision Score" is appropriate here and good accuracy should also be maintained.

Model Building is the most important step in Machine Learning. We apply various models on our dataset and watch for best results. We have used the following models to train our dataset:

- Logistic Regression
- Decision Tree
- Naive Bayes
- Random Forest

First we will see each model one by one.

Logistic Regression

Logistic Regression in Machine Learning

- Logistic regression is one of the most popular Machine Learning algorithms, which comes under the Supervised Learning technique. It is used for predicting the categorical dependent variable using a given set of independent variables.
- Logistic regression predicts the output of a categorical dependent variable. Therefore the outcome must be a categorical or discrete value. It can be either Yes or No, 0 or 1, true or False, etc. but instead of giving the exact value as 0 and 1, it gives the probabilistic values which lie between 0 and 1.
- Logistic Regression is much similar to the Linear Regression except that how they are used. Linear Regression is used for solving Regression problems, whereas Logistic regression is used for solving the classification problems.
- In Logistic regression, instead of fitting a regression line, we fit an "S" shaped logistic function, which predicts two maximum values (0 or 1).
- The curve from the logistic function indicates the likelihood of something such as whether the cells are cancerous or not, a mouse is obese or not based on its weight, etc.
- Logistic Regression is a significant machine learning algorithm because it has the ability to provide probabilities and classify new data using continuous and discrete datasets.
- Logistic Regression can be used to classify the observations using different types of data and can easily determine the most effective variables used for the classification.

Assumptions for Logistic Regression:

- The dependent variable must be categorical in nature.
- The independent variable should not have multi-collinearity.

Type of Logistic Regression:

In our problem statement, it is mentioned that this dataset has no ordinal type of data.

- **Ordinal:** In ordinal Logistic regression, there can be 3 or more possible ordered types of dependent variables, such as "low", "Medium", or "High".

And here The classification problem is binomial type.

- **Binomial:** In binomial Logistic regression, there can be only two possible types of the dependent variables, such as 0 or 1, Pass or Fail, etc.

Steps in Logistic Regression:

To implement the Logistic Regression using Python, we will use the following steps.

- Data Pre-processing step
- Fitting Logistic Regression to the Training set
- Predicting the test result
- Test accuracy of the result(Creation of Confusion matrix)
- Visualizing the test set result.

1.Data Pre-processing step:

In this step, we will pre-process/prepare the data so that we can use it in our code efficiently.

First of all, we have to import different python module and libraries(numpy, pandas, matplotlib) and thereafter our csv train data file.

Then we will extract the dependent and independent variables from the given dataset.

Now we will split the dataset into a training set and test set.

```
1. # Splitting the dataset into training and test set.
2. from sklearn.model_selection import train_test_split
```

In logistic regression, we will do feature scaling because we want accurate result of predictions. Here we will only scale the independent variable because dependent variable have only 0 and 1 values. Below is the code for it:

```
1. #feature Scaling
2. from sklearn.preprocessing import StandardScaler
3. st_x= StandardScaler()
4. x_train= st_x.fit_transform(x_train)
5. x_test= st_x.transform(x_test)
```

3. Fitting Logistic Regression to the Training set:

We have well prepared our dataset, and now we will train the dataset using the training set. For providing training or fitting the model to the training set, we will import the **LogisticRegression** class of the **sklearn** library.

After importing the class, we will create a classifier object and use it to fit the model to the logistic regression. Below is the code for it:

```
1. #Fitting Logistic Regression to the training set
2. from sklearn.linear_model import LogisticRegression
```

3. classifier= LogisticRegression(random_state=0)
4. classifier.fit(x_train, y_train)

   Hence our model is well fitted to the training set.

4. Predicting the Test Result

Our model is well trained on the training set, so we will now predict the result by using test set data. Below is the code for it:

1. #Predicting the test set result
2. y_pred= classifier.predict(x_test)

Test Accuracy of the result

Now we will create the confusion matrix here to check the accuracy of the classification. To create it, we need to import the **confusion_matrix** function of the sklearn library. After importing the function, we will call it using a new variable **cm**. The function takes two parameters, mainly **y_true**(the actual values) and **y_pred** (the targeted value return by the classifier). Below is the code for it:

1. #Creating the Confusion matrix
2. from sklearn.metrics **import** confusion_matrix
3. cm= confusion_matrix()

5. **Visualizing the training set result**

The goal of the classifier:

We have successfully visualized the training set result for the logistic regression, and our goal for this classification is to divide the users who purchased the SUV car and who did not purchase the car. So from the output graph, we can clearly see the two regions (Purple and Green) with the observation points. The Purple region is for those users who didn't buy the car, and Green Region is for those users who purchased the car.

Linear Classifier:

As we can see from the graph, the classifier is a Straight line or linear in nature as we have used the Linear model for Logistic Regression. In further topics, we will learn for non-linear Classifiers.

Visualizing the test set result:

Our model is well trained using the training dataset. Now, we will visualize the result for new observations (Test set). The code for the test set will remain same as above except that here we will use x_test and y_test instead of x_train and y_train**.**

For logistic regression model we need linear_model from sklearn, which we already imported. Next we create a object of Logistic Regression model for prediction.

Now we fit our model by taking the already selected 26 columns and Xtrain and ytrain dataset. Now to see the scores of our model we take trainp and testp by prediction from our model object and by passing Xtrain and Xtest respectively.

And then by using them we print out Accuracy, Precision, Confusion matrix, AUC, F1 score of train and test prediction.

Code→

```
model=linear_model.LogisticRegression()

model.fit(Xtrain[cols],ytrain)

trainp=model.predict(Xtrain[cols])

testp=model.predict(Xtest[cols])

print("Accuracy of training",metrics.accuracy_score(ytrain,trainp))

print("Accuracy of test",metrics.accuracy_score(ytest,testp))

print("Precision of training",metrics.precision_score(ytrain,trainp))

print("Precision of test",metrics.precision_score(ytest,testp))

print("Confusion matrix of training",metrics.confusion_matrix(ytrain,trainp))

print("Confusion matrix of test",metrics.confusion_matrix(ytest,testp))

print("AUC of training",metrics.roc_auc_score(ytrain,trainp))

print("AUC of test",metrics.roc_auc_score(ytest,testp))

print("F1 score of training",metrics.f1_score(ytrain,trainp))

print("F1 score of test",metrics.f1_score(ytest,testp))
```

```
Accuracy of training 0.804277466759972
Accuracy of test 0.7985130111524164
Precision of training 0.8191838244359512
Precision of test 0.8146555983792865
Confusion matrix of training [[ 7112 14666]
 [ 3234 66444]]
Confusion matrix of test [[ 1771  3751]
 [  856 16487]]
AUC of training 0.6400772970671056
AUC of test 0.6356800210792648
F1 score of training 0.8812902883518583
F1 score of test 0.8774114579175647
```

Decision Tree

Decision Tree Classification Algorithm

- Decision Tree is a Supervised learning technique that can be used for both classification and Regression problems, but mostly it is preferred for solving Classification problems. It is a tree-structured classifier, where internal nodes represent the features of a dataset, branches represent the decision rules and each leaf node represents the outcome.
- In a Decision tree, there are two nodes, which are the Decision Node and Leaf Node. Decision nodes are used to make any decision and have multiple branches, whereas Leaf nodes are the output of those decisions and do not contain any further branches.
- The decisions or the test are performed on the basis of features of the given dataset.
- It is a graphical representation for getting all the possible solutions to a problem/decision based on given conditions.
- It is called a decision tree because, similar to a tree, it starts with the root node, which expands on further branches and constructs a tree-like structure.
- In order to build a tree, we use the CART algorithm, which stands for Classification and Regression Tree algorithm.
- A decision tree simply asks a question, and based on the answer (Yes/No), it further split the tree into subtrees.

Steps in Decision Tree Classification:

- Data Pre-processing step
- Fitting a Decision-Tree algorithm to the Training set
- Predicting the test result
- Test accuracy of the result(Creation of Confusion matrix)
- Visualizing the test set result.

1. Data Pre-Processing Step:

```
1. # importing libraries
2. import numpy as nm
3. import matplotlib.pyplot as mtp
4. import pandas as pd
5.
6. #importing datasets
7. data_set= pd.read_csv('user_data.csv')
8.
```

```
#Extracting Independent and dependent Variable
x= data_set.iloc[:, [2,3]].values
y= data_set.iloc[:, 4].values

# Splitting the dataset into training and test set.
from sklearn.model_selection import train_test_split
x_train, x_test, y_train, y_test= train_test_split(x, y, test_size= 0.25, random_state=0)

#feature Scaling
from sklearn.preprocessing import StandardScaler
st_x= StandardScaler()
x_train= st_x.fit_transform(x_train)
x_test= st_x.transform(x_test)
```

2. Fitting a Decision-Tree algorithm to the Training set

Now we will fit the model to the training set. For this, we will import the DecisionTreeClassifier class from sklearn.tree library. Below is the code for it:

```
#Fitting Decision Tree classifier to the training set
From sklearn.tree import DecisionTreeClassifier
classifier= DecisionTreeClassifier(criterion='entropy', random_state=0)
classifier.fit(x_train, y_train)
```

In the above code, we have created a classifier object, in which we have passed two main parameters;

- "criterion='entropy': Criterion is used to measure the quality of split, which is calculated by information gain given by entropy.
- random_state=0": For generating the random states.

3. Predicting the test result

Now we will predict the test set result. We will create a new prediction vector y_pred. Below is the code for it:

```
#Predicting the test set result
y_pred= classifier.predict(x_test)
```

Output:

In the below output image, the predicted output and real test output are given. We can clearly see that there are some values in the prediction vector, which are different from the real vector values. These are prediction errors.

4. Test accuracy of the result (Creation of Confusion matrix)

In the above output, we have seen that there were some incorrect predictions, so if we want to know the number of correct and incorrect predictions, we need to use the confusion matrix. Below is the code for it:

```
1. #Creating the Confusion matrix
2. from sklearn.metrics import confusion_matrix
3. cm= confusion_matrix(y_test, y_pred)
```

Output:

In the above output image, we can see the confusion matrix, which has **6+3= 9 incorrect predictions** and**62+29=91 correct predictions. Therefore, we can say that compared to other classification models, the Decision Tree classifier made a good prediction.**

As we can see in the above image that there are some green data points within the purple region and vice versa. So, these are the incorrect predictions which we have discussed in the confusion matrix.

For Decision Tree model we need tree from sklearn, which we already imported. Next we create a object of Decision Tree model for prediction.

As our train data is very large so we can't apply GridSearchCV for hyperparameter tuning. So we randomly taken hyperparameters and compare results manually and finally we decided to go with : max_depth=10,min_samples_split=27,min_samples_leaf=22,random_state=100.

Now we fit our model by taking the already selected 26 columns and Xtrain and ytrain dataset based on 3 hyperparameters. Now to see the scores of our model we take trainp and testp by prediction from our model object and by passing Xtrain and Xtest respectively.

And then by using them we print out Accuracy, Precision, Confusion matrix, AUC, F1 score of train and test prediction.

Code→

```
model=tree.DecisionTreeClassifier(max_depth=10,min_samples_split=27,min_samples_leaf=22,random_state=100)

model.fit(Xtrain[cols],ytrain)

trainp=model.predict(Xtrain[cols])

testp=model.predict(Xtest[cols])

print("Accuracy of training",metrics.accuracy_score(ytrain,trainp))

print("Accuracy of test",metrics.accuracy_score(ytest,testp))

print("Precision of training",metrics.precision_score(ytrain,trainp))

print("Precision of test",metrics.precision_score(ytest,testp))
```

```
print("Confusion matrix of training",metrics.confusion_matrix(ytrain,trainp))

print("Confusion matrix of test",metrics.confusion_matrix(ytest,testp))

print("AUC of training",metrics.roc_auc_score(ytrain,trainp))

print("AUC of test",metrics.roc_auc_score(ytest,testp))

print("F1 score of training",metrics.f1_score(ytrain,trainp))

print("F1 score of test",metrics.f1_score(ytest,testp))
```

```
Accuracy of training 0.8955453988803359
Accuracy of test 0.8140826590859392
Precision of training 0.8966081346719613
Precision of test 0.8293419199034011
Confusion matrix of training [[13941  7837]
 [ 1716 67962]]
Confusion matrix of test [[ 2130  3392]
 [  859 16484]]
AUC of training 0.807756927290144
AUC of test 0.668099869135891
F1 score of training 0.934333262302632
F1 score of test 0.8857841425078589
```

Naive Bayes

Naïve Bayes Classifier Algorithm

- Naïve Bayes algorithm is a supervised learning algorithm, which is based on Bayes theorem and used for solving classification problems.
- It is mainly used in *text classification* that includes a high-dimensional training dataset.
- Naïve Bayes Classifier is one of the simple and most effective Classification algorithms which helps in building the fast machine learning models that can make quick predictions.
- It is a probabilistic classifier, which means it predicts on the basis of the probability of an object.
- Some popular examples of Naïve Bayes Algorithm are spam filtration, Sentimental analysis, and classifying articles.

Why is it called Naïve Bayes?

The Naïve Bayes algorithm is comprised of two words Naïve and Bayes, Which can be described as:

- **Naïve**: It is called Naïve because it assumes that the occurrence of a certain feature is independent of the occurrence of other features. Such as if the fruit is identified on the bases of color, shape, and taste, then red, spherical, and sweet fruit is recognized as an apple. Hence each feature individually contributes to identify that it is an apple without depending on each other.
- **Bayes**: It is called Bayes because it depends on the principle of Bayes' Theorem.

Bayes' Theorem:

- Bayes' theorem is also known as Bayes' Rule or Bayes' law, which is used to determine the probability of a hypothesis with prior knowledge. It depends on the conditional probability.
- The formula for Bayes' theorem is given as:

$$P(A|B)= \frac{P(B|A)P(A)}{P(B)}$$

Where,

P(A|B) is Posterior probability: Probability of hypothesis A on the observed event B.

P(B|A) is Likelihood probability: Probability of the evidence given that the probability of a hypothesis is true

P(A) is Prior Probability: Probability of hypothesis before observing the evidence.

P(B) is Marginal Probability: Probability of Evidence.

Steps to implement:

- Data Pre-processing step
- Fitting Naive Bayes to the Training set
- Predicting the test result
- Test accuracy of the result(Creation of Confusion matrix)
- Visualizing the test set result.

1) Data Pre-processing step:

In this step, we will pre-process/prepare the data so that we can use it efficiently in our code. It is similar as we did in data-pre-processing.

2) Fitting Naive Bayes to the Training Set:

After the pre-processing step, now we will fit the Naive Bayes model to the Training set. Below is the code for it:

```
1. # Fitting Naive Bayes to the Training set
2. from sklearn.naive_bayes import GaussianNB
3. classifier = GaussianNB()
4. classifier.fit(x_train, y_train)
```

In the above code, we have used the GaussianNB classifier to fit it to the training dataset. We can also use other classifiers as per our requirement.

3) Prediction of the test set result:

Now we will predict the test set result. For this, we will create a new predictor variable y_pred, and will use the predict function to make the predictions.

```
1. # Predicting the Test set results
2. y_pred = classifier.predict(x_test)
```

The above output shows the result for prediction vector y_pred and real vector y_test. We can see that some predications are different from the real values, which are the incorrect predictions.

4) Creating Confusion Matrix:

Now we will check the accuracy of the Naive Bayes classifier using the Confusion matrix. Below is the code for it:

```
1. # Making the Confusion Matrix
2. from sklearn.metrics import confusion_matrix
3. cm = confusion_matrix(y_test, y_pred)
```

4. The above output is final output for test set data. As we can see the classifier has created a Gaussian curve to divide the "purchased" and "not purchased" variables. There are some wrong predictions which we have calculated in Confusion matrix. But still it is pretty good classifier.

For Naive Bayes model we need naive_bayes from sklearn, which we already imported. Next we create a object of Naive Bayes model for prediction.

Now we fit our model by taking the already selected 26 columns and Xtrain and ytrain dataset. Now to see the scores of our model we take trainp and testp by prediction from our model object and by passing Xtrain and Xtest respectively.

And then by using them we print out Accuracy, Precision, Confusion matrix, AUC, F1 score of train and test prediction.

Code→

```
model=naive_bayes.GaussianNB()

model.fit(Xtrain[cols],ytrain)

trainp=model.predict(Xtrain[cols])

testp=model.predict(Xtest[cols])

print("Accuracy of training",metrics.accuracy_score(ytrain,trainp))

print("Accuracy of test",metrics.accuracy_score(ytest,testp))

print("Precision of training",metrics.precision_score(ytrain,trainp))

print("Precision of test",metrics.precision_score(ytest,testp))

print("Confusion matrix of training",metrics.confusion_matrix(ytrain,trainp))

print("Confusion matrix of test",metrics.confusion_matrix(ytest,testp))

print("AUC of training",metrics.roc_auc_score(ytrain,trainp))

print("AUC of test",metrics.roc_auc_score(ytest,testp))

print("F1 score of training",metrics.f1_score(ytrain,trainp))

print("F1 score of test",metrics.f1_score(ytest,testp))
```

```
Accuracy of training 0.602103743876837
Accuracy of test 0.5973321670675705
Precision of training 0.8815156099573649
Precision of test 0.8817567567567568
Confusion matrix of training [[16609  5169]
 [31221 38457]]
Confusion matrix of test [[4262 1260]
 [7947 9396]]
AUC of training 0.6572874742069162
AUC of test 0.6567982915721179
F1 score of training 0.6788286380004236
F1 score of test 0.6711668273866923
```

Random Forest

Random Forest Algorithm:

Random Forest is a popular machine learning algorithm that belongs to the supervised learning technique. It can be used for both Classification and Regression problems in ML. It is based on the concept of ensemble learning, which is a process of *combining multiple classifiers to solve a complex problem and to improve the performance of the model.*

As the name suggests, "Random Forest is a classifier that contains a number of decision trees on various subsets of the given dataset and takes the average to improve the predictive accuracy of that dataset." Instead of relying on one decision tree, the random forest takes the prediction from each tree and based on the majority votes of predictions, and it predicts the final output.

The greater number of trees in the forest leads to higher accuracy and prevents the problem of overfitting.

Assumptions for Random Forest

Since the random forest combines multiple trees to predict the class of the dataset, it is possible that some decision trees may predict the correct output, while others may not. But together, all the trees predict the correct output. Therefore, below are two assumptions for a better Random forest classifier:

- There should be some actual values in the feature variable of the dataset so that the classifier can predict accurate results rather than a guessed result.
- The predictions from each tree must have very low correlations.

Python Implementation of Random Forest Algorithm:

Implementation Steps are given below:

- Data Pre-processing step
- Fitting the Random forest algorithm to the Training set
- Predicting the test result
- Test accuracy of the result (Creation of Confusion matrix)
- Visualizing the test set result.

1.Data Pre-Processing Step:

Below is the code for the pre-processing step:

```
1. # importing libraries
2. import numpy as nm
3. import matplotlib.pyplot as mtp
4. import pandas as pd
5.
6. #importing datasets
```

```
7. data_set= pd.read_csv('user_data.csv')
8.
9. #Extracting Independent and dependent Variable
10. x= data_set.iloc[:, [2,3]].values
11. y= data_set.iloc[:, 4].values
12.
13. # Splitting the dataset into training and test set.
14. from sklearn.model_selection import train_test_split
15. x_train, x_test, y_train, y_test= train_test_split(x, y, test_size= 0.25, random_state=0)
16.
17. #feature Scaling
18. from sklearn.preprocessing import StandardScaler
19. st_x= StandardScaler()
20. x_train= st_x.fit_transform(x_train)
21. x_test= st_x.transform(x_test)
22.
```

2. Fitting the Random Forest algorithm to the training set:

Now we will fit the Random forest algorithm to the training set. To fit it, we will import the RandomForestClassifie**r** class from the sklearn.ensemble library. The code is given below:

```
1. #Fitting Decision Tree classifier to the training set
2. from sklearn.ensemble import RandomForestClassifier
3. classifier= RandomForestClassifier(n_estimators= 10, criterion="entropy")
4. classifier.fit(x_train, y_train)
```

In the above code, the classifier object takes below parameters:

- **n_estimators=** The required number of trees in the Random Forest. The default value is 10. We can choose any number but need to take care of the overfitting issue.
- **criterion=** It is a function to analyze the accuracy of the split. Here we have taken "entropy" for the information gain.

Output:

3. Predicting the Test Set result

Since our model is fitted to the training set, so now we can predict the test result. For prediction, we will create a new prediction vector y_pred. Below is the code for it:

```
1. #Predicting the test set result
2. y_pred= classifier.predict(x_test)
```

Output:

By checking the above prediction vector and test set real vector, we can determine the incorrect predictions done by the classifier.

4. Creating the Confusion Matrix

Now we will create the confusion matrix to determine the correct and incorrect predictions. Below is the code for it:

```
1. #Creating the Confusion matrix
2. from sklearn.metrics import confusion_matrix
3. cm= confusion_matrix(y_test, y_pred)
```

output:

Output:

The above image is the visualization result for the test set. We can check that there is a minimum number of incorrect predictions (8) without the Overfitting issue. We will get different results by changing the number of trees in the classifier.

For Random Forest model we need ensemble from sklearn, which we already imported. Next we create a object of Random Forest model for prediction.

Now we fit our model by taking the already selected 26 columns and Xtrain and ytrain dataset. Now to see the scores of our model we take trainp and testp by prediction from our model object and by passing Xtrain and Xtest respectively.

And then by using them we print out Accuracy, Precision, Confusion matrix, AUC, F1 score of train and test prediction.

Code→

```
model=ensemble.RandomForestClassifier()

model.fit(Xtrain[cols],ytrain)

trainp=model.predict(Xtrain[cols])

testp=model.predict(Xtest[cols])

print("Accuracy of training",metrics.accuracy_score(ytrain,trainp))

print("Accuracy of test",metrics.accuracy_score(ytest,testp))

print("Precision of training",metrics.precision_score(ytrain,trainp))

print("Precision of test",metrics.precision_score(ytest,testp))

print("Confusion matrix of training",metrics.confusion_matrix(ytrain,trainp))

print("Confusion matrix of test",metrics.confusion_matrix(ytest,testp))

print("AUC of training",metrics.roc_auc_score(ytrain,trainp))
```

```
print("AUC of test",metrics.roc_auc_score(ytest,testp))

print("F1 score of training",metrics.f1_score(ytrain,trainp))

print("F1 score of test",metrics.f1_score(ytest,testp))
```

```
Accuracy of training 1.0
Accuracy of test 0.8233107369341789
Precision of training 1.0
Precision of test 0.825974025974026
Confusion matrix of training [[21778     0]
 [    0 69678]]
Confusion matrix of test [[ 1971  3551]
 [  489 16854]]
AUC of training 1.0
AUC of test 0.6643700394597172
F1 score of training 1.0
F1 score of test 0.892974462223164
```

Now we write a function to get all the scores of 4 model used in a compact way

By using the below function we find all the scores within a single output.

Code→

```
from sklearn import tree
from sklearn import neighbors
from sklearn import naive_bayes
from sklearn import ensemble


def modelstats1(Xtrain,Xtest,ytrain,ytest):
   stats=[]
   modelnames=["LR","DecisionTree","NB","RF"]#,"KNN"
   models=list()
   models.append(linear_model.LogisticRegression())

models.append(tree.DecisionTreeClassifier(max_depth=10,min_samples_split=27,min_samples_leaf=22,random_state=100))
   #models.append(neighbors.KNeighborsClassifier())
   models.append(naive_bayes.GaussianNB())
   models.append(ensemble.RandomForestClassifier())
   for name,model in zip(modelnames,models):
      if name=="KNN":
         k=[l for l in range(5,17,2)]
         grid={"n_neighbors":k}
         grid_obj=model_selection.GridSearchCV(estimator=model,param_grid=grid,scoring="f1")
         grid_fit=grid_obj.fit(Xtrain,ytrain)
         model =grid_fit.best_estimator_
         model.fit(Xtrain,ytrain)
         name=name+"("+str(grid_fit.best_params_["n_neighbors"])+")"
      else:
         model.fit(Xtrain,ytrain)
         trainprediction=model.predict(Xtrain)
         testprediction=model.predict(Xtest)
```

```
        scores=list()
        scores.append(name+"-train")
        scores.append(metrics.accuracy_score(ytrain,trainprediction))
        scores.append(metrics.precision_score(ytrain,trainprediction))
        scores.append(metrics.recall_score(ytrain,trainprediction))
        scores.append(metrics.roc_auc_score(ytrain,trainprediction))
        scores.append(metrics.f1_score(ytrain,trainprediction))
        stats.append(scores)
        scores=list()
        scores.append(name+"-test")
        scores.append(metrics.accuracy_score(ytest,testprediction))
        scores.append(metrics.precision_score(ytest,testprediction))
        scores.append(metrics.recall_score(ytest,testprediction))
        scores.append(metrics.roc_auc_score(ytest,testprediction))
        scores.append(metrics.f1_score(ytest,testprediction))
        stats.append(scores)
    colnames=["MODELNAME","ACCURACY","PRECISION","RECALL","AUC","F1_SCORE"]
    return pd.DataFrame(stats,columns=colnames)


modelstats1(Xtrain[cols],Xtest[cols],ytrain,ytest) #calling function
```

	MODELNAME	ACCURACY	PRECISION	RECALL	AUC	F1_SCORE
0	LR-train	0.804277	0.819184	0.953586	0.640077	0.881290
1	LR-test	0.798513	0.814656	0.950643	0.635680	0.877411
2	DecisionTree-train	0.895545	0.896608	0.975372	0.807757	0.934333
3	DecisionTree-test	0.814083	0.829342	0.950470	0.668100	0.885784
4	NB-train	0.602104	0.881516	0.551925	0.657287	0.678829
5	NB-test	0.597332	0.881757	0.541775	0.656798	0.671167
6	RF-train	1.000000	1.000000	1.000000	1.000000	1.000000
7	RF-test	0.823311	0.825974	0.971804	0.664370	0.892974

Above table represents scores of various models used.

Now by observing the scores we can see that the Random Forest model is overfitted.

Naïve Bayes Model gives the worse result.

And Decision Tree model gives the best result.

So the ML algorithm that performs the best was Decision Tree model with AUC 0.668 and f1 score 0.885

Conclusion:

So, the ML algorithm that performs the best was Decision Tree model with precision score 0.829, AUC 0.668 and f1__score 0.885 with 26 selected features.

The 26 selected columns that are the part of model are 'v12', 'v16', 'v19', 'v21', 'v22', 'v24', 'v28', 'v30', 'v36', 'v40', 'v50', 'v52', 'v56', 'v62', 'v66', 'v70', 'v72', 'v75', 'v78', 'v79', 'v112', 'v113', 'v117', 'v123', 'v125', 'v129'.

Code

Project Link:

Repository Link : https://github.com/Shreya9Mallick/Project

Jupyter Notebook Code:

import some helpful packages

In [ ]:

```
# import some helpful packages
import numpy as np      # linear algebra
import pandas as pd       # data processing, CSV file I/O
import matplotlib.pyplot as plt      # for plotting data
import seaborn as sns      # used for statistical graph
from sklearn import model_selection  # for train test split
from sklearn import linear_model     # for model object
```

reading train data set and explore

In [ ]:

```
# reading train data
df=pd.read_csv("train.csv")
print(df.shape)
```

In [ ]:

df[:5] *# view first 5 rows of data set*

In [ ]:

```
df.drop("ID",axis=1,inplace=True)      # ID is not required for our work
```

In [ ]:

```
df[:5]     # again view first 5 rows of data set
```

In [ ]:

```
df.describe()     # see the data range
```

In [ ]:

```
df.describe(include="object")     # string valued columns
```

In [ ]:

```
df.describe(include="int64")    # integer columns
```

In [ ]:

```
df.describe(include="float64")    # floating valued columns
```

Null value analysis

In [ ]:

```
nullcol = list()
print("Number of columns with null value:",end=" ")
for col in df:
    if(df[col].isna().sum())!=0:
        nullcol.append(col)
print(len(nullcol),end="\n\n")
print("Number of columns with no null value:",end=" ")
nonnullcol = list()
for col in df:
    if(df[col].isna().sum())==0:
        nonnullcol.append(col)
print(len(nonnullcol),end="\n\n")
print("Columns with null value(percentage):")
for col in df:
    if(df[col].isna().sum())!=0:
        print(col,round(((df[col].isna().sum())/df.shape[0])*100,2),end=" // ")
```

In [ ]:

```
print("columns with no null value:",*nonnullcol)   # so 4 integer columns have no null
value and some other string coluns also
df[nonnullcol][:10]
```

In [ ]:

```
# ploting correlation of columns(v1 to v26) in heatmap to see correlation between two
column

cols = list()
for i in range(1,27):
    a = "v"+str(i)
    cols.append(a)
df_corr = df[cols]
corr=df_corr.corr()
sns.set(font_scale=1.25)
plt.figure(figsize=(20,20))
```

```
hm = sns.heatmap(corr, cbar=True, annot=True, square=True, fmt='.2f',
annot_kws={'size': 20}, yticklabels=corr.columns.values,
xticklabels=corr.columns.values, linecolor='white')
plt.show()
```

In [ ]:

```
# ploting correlation of columns(v27 to v52) in heatmap to see correlation between two
column

cols = list()
for i in range(27,53):
    a = "v"+str(i)
    cols.append(a)
df_corr = df[cols]
corr=df_corr.corr()
sns.set(font_scale=1.25)
plt.figure(figsize=(20,20))
hm = sns.heatmap(corr, cbar=True, annot=True, square=True, fmt='.2f',
annot_kws={'size': 20}, yticklabels=corr.columns.values,
xticklabels=corr.columns.values, linecolor='white')
plt.show()
```

In [ ]:

```
# ploting correlation of columns(v53 to v78) in heatmap to see correlation between two
column

cols = list()
for i in range(53,79):
    a = "v"+str(i)
    cols.append(a)
df_corr = df[cols]
corr=df_corr.corr()
sns.set(font_scale=1.25)
plt.figure(figsize=(20,20))
hm = sns.heatmap(corr, cbar=True, annot=True, square=True, fmt='.2f',
annot_kws={'size': 20}, yticklabels=corr.columns.values,
xticklabels=corr.columns.values, linecolor='white')
plt.show()
```

In [ ]:

```
# ploting correlation of columns(v79 to v104) in heatmap to see correlation between two
column

cols = list()
for i in range(79,105):
    a = "v"+str(i)
    cols.append(a)
df_corr = df[cols]
corr=df_corr.corr()
sns.set(font_scale=1.25)
plt.figure(figsize=(20,20))
```

```
hm = sns.heatmap(corr, cbar=True, annot=True, square=True, fmt='.2f',
annot_kws={'size': 20}, yticklabels=corr.columns.values,
xticklabels=corr.columns.values, linecolor='white')
plt.show()
```

In [ ]:

```
# ploting correlation of columns(v105 to v131) in heatmap to see correlation between
two column

cols = list()
for i in range(105,132):
    a = "v"+str(i)
    cols.append(a)
df_corr = df[cols]
corr=df_corr.corr()
sns.set(font_scale=1.25)
plt.figure(figsize=(20,20))
hm = sns.heatmap(corr, cbar=True, annot=True, square=True, fmt='.2f',
annot_kws={'size': 20}, yticklabels=corr.columns.values,
xticklabels=corr.columns.values, linecolor='white')
plt.show()
```

In [ ]:

```
# plot of null value counts of columns to see the distribution of null values

print("Null value count graph")
missing = df.isnull().sum()
missing = missing[missing > 0]
missing.sort_values(inplace=True)
missing.plot.bar(figsize=(15,6))
plt.show()
```

In [ ]:

```
uselesscol = list()
for col in df:
    if(df[col].isna().sum())!=0:
        if ((df[col].isna().sum())/df.shape[0])*100>30:
            uselesscol.append(col)
print("There are ",len(uselesscol),"columns where null value is more than 30%")
```

Unique value count of each columns

In [ ]:

```
objcols=df.select_dtypes(include="object").columns
floatcols=df.select_dtypes(include="float64").columns
```

```
intcols=df.select_dtypes(include="int64").columns
```

In [ ]:

```
print("Number of object columns:",objcols.shape[0])
print("Unique value count of each object column:")
for col in df[objcols]:
    print(col," : ",df[col].nunique())
```

In [ ]:

```
print("Number of floating columns:",floatcols.shape[0])
print("Unique value count of each float64 column:")
for col in df[floatcols]:
    print(col," : ",df[col].nunique())
```

In [ ]:

```
print("Number of integer columns:",intcols.shape[0])
print("Unique value count of each integer column:")
for col in df[intcols]:
    print(col," : ",df[col].nunique())
```

In [ ]:

```
sns.countplot(x="target",data=df)   # distribution of target value
```

In [ ]:

```
df["target"].value_counts()
```

Exploratory data analysis

In [ ]:

```
# countplot of categorical column to see the frequency of each catagorical data
sns.set_style("dark")
for cols in objcols:
    if cols=="v22":
        continue
    plt.figure(figsize=(15,4))
    plt.subplot(121)
```

```
    sns.countplot(x=cols,data=df)
    plt.subplot(122)
    sns.countplot(x=cols,hue="target",data=df)
    plt.tight_layout()
    plt.show()
```

In [ ]:

```
# countplot of integer(discrete) column to see the frequency of each catagorical data
sns.set_style("dark")
for cols in intcols[2:]:
    plt.figure(figsize=(15,4))
    plt.subplot(121)
    sns.countplot(x=cols,data=df)
    plt.subplot(122)
    sns.countplot(x=cols,hue="target",data=df)
    plt.tight_layout()
    plt.show()
```

In [ ]:

```
# box plot of floating columns to see the distribution of data

sns.set_style("dark")
for col in floatcols:
    plt.figure(figsize=(15,4))
    plt.subplot(121)
    sns.distplot(df[col],label="skew: "+str(np.round(df[col].skew(),2)))
    plt.legend()
    plt.subplot(122)
    sns.boxplot(y=col,x="target",data=df)
    plt.tight_layout()
    plt.show()

print(col,"Min:",df[col].min(),"Q1:",df[col].quantile(0.25),"Q2:",df[col].quantile(0.5)
,"Q3:",df[col].quantile(0.75),"Max:",df[col].max())
```

Data Cleaning

In [ ]:

```
pd.crosstab(index=df["v3"],columns=df["target"],normalize="columns")    # A and B value
counts are very less compare to C in v3 column
```

In [ ]:

```
# column "v3" is dropped

df.drop("v3",axis=1,inplace=True)
print(df.shape)
```

In [ ]:

```
pd.crosstab(index=df["v38"],columns=df["target"],normalize="columns")    # other value
counts are very less compare to 0 in v38 column
```

In [ ]:

```
# column "v38" is dropped

df.drop("v38",axis=1,inplace=True)
print(df.shape)
```

In [ ]:

```
pd.crosstab(index=df["v74"],columns=df["target"],normalize="columns")    # A and C
value counts are very less compare to B in v74 column
```

In [ ]:

```
# column "v74" is dropped

df.drop("v74",axis=1,inplace=True)
print(df.shape)
```

Null value handling

In [ ]:

```
# fill NaN values with mean

for f in df.columns:
    if df[f].dtype == 'float64':
        m = df.groupby("target")[f].mean()
        df.loc[df.target==0,f]=df.loc[df.target==0,f].fillna(m[0])  # target=0 rows
null value replaced with mean of all target=0 non null rows
        df.loc[df.target==1,f]=df.loc[df.target==1,f].fillna(m[1])  # target=0 rows
null value replaced with mean of all target=0 non null rows
```

In [ ]:

```
#  now see which columns are left with null values

nulcols = []
print("Columns with null value(percentage):")
for col in df:
    if(df[col].isna().sum())!=0:
        print(col,(((df[col].isna().sum())/df.shape[0])*100))
        nulcols.append(col)
```

In [ ]:

```
df[nulcols][:10]    # all columns with null values are object type
```

In [ ]:

```
#filling "missing" in place of null values

for col in nulcols:
    df[col][df[col] != df[col]] = "missing"
```

In [ ]:

```
df[nulcols][:10]
```

Weight of Evidence value Calculation

In [ ]:

```
# we will be using "Weight of Evidence" value to convert all discrete columns to
continious
# below function takes dataframe and column as input and replace "woe" value in that
column in dataframe itself

def woefun(df,col):
    woedf=pd.crosstab(index=df[col],columns=df["target"])
    woedf.rename(columns={0:"zero",1:"one"},inplace=True)
    s1 = woedf["zero"].sum()
```

```
    s2 = woedf["one"].sum()
    for i in woedf.index:
        if woedf["zero"][i]==0 or woedf["one"][i]==0:
            woedf["zero"][i]=woedf["zero"][i]+1
            woedf["one"][i]=woedf["one"][i]+1
    woedf["woe"]=np.log(woedf["zero"]/s1)-np.log(woedf["one"]/s2)
    d = dict(woedf["woe"])
    df[col].replace(d,inplace=True)
```

In [ ]:

```
# calling function for all null valued columns

for col in nulcols:
    woefun(df,col)
```

In [ ]:

```
df[nulcols][:10]    # objects column (with null values) value changed to woe values
```

In [ ]:

```
intcols_list = ["v62","v72","v129"]

print(intcols_list)    # integer(discrete) columns
```

In [ ]:

```
df[intcols_list][:10]
```

In [ ]:

```
# calling function for integer(discrete) columns

for col in intcols_list:
    woefun(df,col)
```

In [ ]:

```
df[intcols_list][:10]    # integer columns value changed to woe values
```

In [ ]:

```
objcols_list = []
objcols=df.select_dtypes(include="object").columns
for col in objcols:
    objcols_list.append(col)
```

```
print(objcols_list)    # object columns left in the data frame
```

In [ ]:

```
df[objcols_list][:10]
```

In [ ]:

```
# calling function for remaining object columns

for col in objcols_list:
    woefun(df,col)
```

In [ ]:

```
df[objcols_list][:10]    # object columns value changed to woe values
```

In [ ]:

```
df.shape    # we are left with "target" column and 128 more columns
```

Train Test split and Data pre-processing

In [ ]:

```
# differentiating X and y for models

y = df["target"]
X = df.drop("target",axis=1)

# spliting dataset int train and test data

Xtrain,Xtest,ytrain,ytest=model_selection.train_test_split(X,y,test_size=.2,random_stat
e=100)
```

In [ ]:

```
Xtrain.shape    #Xtrain data have 91456 rows
```

In [ ]:

```
Xtest.shape    #Xtest data have 22865 rows
```

In [ ]:

```
# importing library for pre-processing of dataa and creating object of that

from sklearn import preprocessing
scaler=preprocessing.StandardScaler()
```

In [ ]:

```
Xtrain[floatcols]=scaler.fit_transform(Xtrain[floatcols])   # fit-transform data to get
normalized values
Xtest[floatcols]=scaler.fit_transform(Xtest[floatcols])
```

Base model building

In [ ]:

```
model=linear_model.LogisticRegression()    # object of logistic regression model

model.fit(Xtrain,ytrain)    # fitting model

trainp=model.predict(Xtrain)
testp=model.predict(Xtest)
```

In [ ]:

```
from sklearn import metrics  # to show the scores of model build

print("Accuracy of training",metrics.accuracy_score(ytrain,trainp))

print("Accuracy of test",metrics.accuracy_score(ytest,testp))

print("Precision of training",metrics.precision_score(ytrain,trainp))

print("Precision of test",metrics.precision_score(ytest,testp))

print("Confusion matrix of training",metrics.confusion_matrix(ytrain,trainp))

print("Confusion matrix of test",metrics.confusion_matrix(ytest,testp))

print("AUC of training",metrics.roc_auc_score(ytrain,trainp))

print("AUC of test",metrics.roc_auc_score(ytest,testp))

print("F1 score of training",metrics.f1_score(ytrain,trainp))

print("F1 score of test",metrics.f1_score(ytest,testp))
```

Recursive Feature Elimination

In [ ]:

```
# Recursive feature elimination to reduce features to 60

from sklearn import feature_selection   # importing required library

logreg=linear_model.LogisticRegression()
rfeobj=feature_selection.RFE(logreg,60)    # creating object to fit model
```

In [ ]:

```
rfeobj.fit(Xtrain,ytrain)    # fitting model to get reduced feature
```

In [ ]:

```
cols=Xtrain.columns[rfeobj.support_]    # taking the 60 selected columns in cols
```

In [ ]:

```
cols
```

In [ ]:

```
list(zip(Xtrain.columns,rfeobj.support_,rfeobj.ranking_))
```

In [ ]:

```
# again we are testing the result with above 60 columns

model=linear_model.LogisticRegression()

model.fit(Xtrain[cols],ytrain)

trainp=model.predict(Xtrain[cols])
testp=model.predict(Xtest[cols])

print("Accuracy of training",metrics.accuracy_score(ytrain,trainp))

print("Accuracy of test",metrics.accuracy_score(ytest,testp))

print("Precision of training",metrics.precision_score(ytrain,trainp))

print("Precision of test",metrics.precision_score(ytest,testp))

print("Confusion matrix of training",metrics.confusion_matrix(ytrain,trainp))

print("Confusion matrix of test",metrics.confusion_matrix(ytest,testp))
```

```
print("AUC of training",metrics.roc_auc_score(ytrain,trainp))

print("AUC of test",metrics.roc_auc_score(ytest,testp))

print("F1 score of training",metrics.f1_score(ytrain,trainp))

print("F1 score of test",metrics.f1_score(ytest,testp))

# the scores remain almost same
```

Variance Inflation Factor

In [ ]:

```
# finding out "Variance Inflation Factor" to further discard some columns

import statsmodels.api as sm
from statsmodels.stats.outliers_influence import variance_inflation_factor
```

In [ ]:

```
vif = pd.DataFrame()  # blank data frame
vif['Features'] = Xtrain[cols].columns
vif['VIF'] = [variance_inflation_factor(Xtrain[cols].values, i) for i in
range(Xtrain[cols].shape[1])]
vif['VIF'] = round(vif['VIF'], 2)
vif = vif.sort_values(by = "VIF", ascending = False)
```

In [ ]:

```
vif
```

In [ ]:

```
Xtrainsm=sm.add_constant(Xtrain[cols])
logm1 = sm.GLM(ytrain,Xtrainsm, family = sm.families.Binomial())    #
model=linear_model.LogisticRegression()

logm1results=logm1.fit()    # fit() return an result object


logm1results.summary()
```

In [ ]:

```
useless_cols =
["v11","v15","v25","v41","v43","v47","v49","v53","v58","v60","v71","v73","v77","v80","v
84","v88","v89","v91","v97","v100","v111","v116","v121"]
lst = []        # we selected useless columns based on P>|z| values
```

```
for i in cols:
    if i in useless_cols:
        continue
    lst.append(i)
cols = lst

print(len(cols))    # remaining 37 columns
```

In [ ]:

```
# again we are testing the result with above remaining columns

model=linear_model.LogisticRegression()

model.fit(Xtrain[cols],ytrain)

trainp=model.predict(Xtrain[cols])
testp=model.predict(Xtest[cols])

print("Accuracy of training",metrics.accuracy_score(ytrain,trainp))

print("Accuracy of test",metrics.accuracy_score(ytest,testp))

print("Precision of training",metrics.precision_score(ytrain,trainp))

print("Precision of test",metrics.precision_score(ytest,testp))

print("Confusion matrix of training",metrics.confusion_matrix(ytrain,trainp))

print("Confusion matrix of test",metrics.confusion_matrix(ytest,testp))

print("AUC of training",metrics.roc_auc_score(ytrain,trainp))

print("AUC of test",metrics.roc_auc_score(ytest,testp))

print("F1 score of training",metrics.f1_score(ytrain,trainp))

print("F1 score of test",metrics.f1_score(ytest,testp))

# the scores remain almost same
```

In [ ]:

```
vif = pd.DataFrame()  # blank data frame
vif['Features'] = Xtrain[cols].columns
vif['VIF'] = [variance_inflation_factor(Xtrain[cols].values, i) for i in
range(Xtrain[cols].shape[1])]
vif['VIF'] = round(vif['VIF'], 2)
vif = vif.sort_values(by = "VIF", ascending = False)
```

In [ ]:

```
Vif
```

In [ ]:

```
Xtrainsm=sm.add_constant(Xtrain[cols])
logm1 = sm.GLM(ytrain,Xtrainsm, family = sm.families.Binomial())    #
model=linear_model.LogisticRegression()

logm1results=logm1.fit()    # fit() return an result object


logm1results.summary()
```

In [ ]:

```
useless_cols = ["v7","v9","v26","v46","v65","v98","v105"]    # we selected useless
columns based on P>|z| values
lst = []
for i in cols:
    if i in useless_cols:
        continue
    lst.append(i)
cols = lst

print(len(cols))   # now left with 30 columns
```

In [ ]:

```
# again we are testing the result with above remaining columns

model=linear_model.LogisticRegression()

model.fit(Xtrain[cols],ytrain)

trainp=model.predict(Xtrain[cols])
testp=model.predict(Xtest[cols])

print("Accuracy of training",metrics.accuracy_score(ytrain,trainp))

print("Accuracy of test",metrics.accuracy_score(ytest,testp))

print("Precision of training",metrics.precision_score(ytrain,trainp))

print("Precision of test",metrics.precision_score(ytest,testp))

print("Confusion matrix of training",metrics.confusion_matrix(ytrain,trainp))

print("Confusion matrix of test",metrics.confusion_matrix(ytest,testp))

print("AUC of training",metrics.roc_auc_score(ytrain,trainp))

print("AUC of test",metrics.roc_auc_score(ytest,testp))

print("F1 score of training",metrics.f1_score(ytrain,trainp))

print("F1 score of test",metrics.f1_score(ytest,testp))

# the scores remain almost same
```

In [ ]:

```
vif = pd.DataFrame()  # blank data frame
vif['Features'] = Xtrain[cols].columns
vif['VIF'] = [variance_inflation_factor(Xtrain[cols].values, i) for i in
range(Xtrain[cols].shape[1])]
vif['VIF'] = round(vif['VIF'], 2)
vif = vif.sort_values(by = "VIF", ascending = False)
```

In [ ]:

vif

In [ ]:

```
Xtrainsm=sm.add_constant(Xtrain[cols])
logm1 = sm.GLM(ytrain,Xtrainsm, family = sm.families.Binomial())    #
model=linear_model.LogisticRegression()

logm1results=logm1.fit()    # fit() return an result object


logm1results.summary()
```

In [ ]:

```
useless_cols = ["v10","v8","v31","v110"]    # we selected useless columns based on
P>|z| values
lst = []
for i in cols:
    if i in useless_cols:
        continue
    lst.append(i)
cols = lst

print(len(cols))    # remaining 26 columns
```

In [ ]:

```
# again we are testing the result with above remaining 26 columns

model=linear_model.LogisticRegression()

model.fit(Xtrain[cols],ytrain)

trainp=model.predict(Xtrain[cols])
testp=model.predict(Xtest[cols])

print("Accuracy of training",metrics.accuracy_score(ytrain,trainp))

print("Accuracy of test",metrics.accuracy_score(ytest,testp))

print("Precision of training",metrics.precision_score(ytrain,trainp))
```

```
print("Precision of test",metrics.precision_score(ytest,testp))

print("Confusion matrix of training",metrics.confusion_matrix(ytrain,trainp))

print("Confusion matrix of test",metrics.confusion_matrix(ytest,testp))

print("AUC of training",metrics.roc_auc_score(ytrain,trainp))

print("AUC of test",metrics.roc_auc_score(ytest,testp))

print("F1 score of training",metrics.f1_score(ytrain,trainp))

print("F1 score of test",metrics.f1_score(ytest,testp))

# the scores remain almost same
```

In [ ]:

```
vif = pd.DataFrame()  # blank data frame
vif['Features'] = Xtrain[cols].columns
vif['VIF'] = [variance_inflation_factor(Xtrain[cols].values, i) for i in
range(Xtrain[cols].shape[1])]
vif['VIF'] = round(vif['VIF'], 2)
vif = vif.sort_values(by = "VIF", ascending = False)
```

In [ ]:

```
vif    # all vif value are < 5
```

In [ ]:

```
Xtrainsm=sm.add_constant(Xtrain[cols])
logm1 = sm.GLM(ytrain,Xtrainsm, family = sm.families.Binomial())    #
model=linear_model.LogisticRegression()

logm1results=logm1.fit()    # fit() return an result object


logm1results.summary()    # all P>|z| values are < 0.05
```

Final columns for model building

In [ ]:

```
print(len(cols))    # final colums selected for model building
print(cols)
```

Logistic Regression

In [ ]:

```
model=linear_model.LogisticRegression()

model.fit(Xtrain[cols],ytrain)

trainp=model.predict(Xtrain[cols])
testp=model.predict(Xtest[cols])

print("Accuracy of training",metrics.accuracy_score(ytrain,trainp))

print("Accuracy of test",metrics.accuracy_score(ytest,testp))

print("Precision of training",metrics.precision_score(ytrain,trainp))

print("Precision of test",metrics.precision_score(ytest,testp))

print("Confusion matrix of training",metrics.confusion_matrix(ytrain,trainp))

print("Confusion matrix of test",metrics.confusion_matrix(ytest,testp))

print("AUC of training",metrics.roc_auc_score(ytrain,trainp))

print("AUC of test",metrics.roc_auc_score(ytest,testp))

print("F1 score of training",metrics.f1_score(ytrain,trainp))

print("F1 score of test",metrics.f1_score(ytest,testp))
```

In [ ]:

```
from sklearn import tree
from sklearn import neighbors
from sklearn import naive_bayes
from sklearn import ensemble
```

Decision Tree

In [ ]:

```
model=tree.DecisionTreeClassifier(max_depth=10,min_samples_split=27,min_samples_leaf=22
,random_state=100)

model.fit(Xtrain[cols],ytrain)

trainp=model.predict(Xtrain[cols])
testp=model.predict(Xtest[cols])

print("Accuracy of training",metrics.accuracy_score(ytrain,trainp))

print("Accuracy of test",metrics.accuracy_score(ytest,testp))
```

```
print("Precision of training",metrics.precision_score(ytrain,trainp))

print("Precision of test",metrics.precision_score(ytest,testp))

print("Confusion matrix of training",metrics.confusion_matrix(ytrain,trainp))

print("Confusion matrix of test",metrics.confusion_matrix(ytest,testp))

print("AUC of training",metrics.roc_auc_score(ytrain,trainp))

print("AUC of test",metrics.roc_auc_score(ytest,testp))

print("F1 score of training",metrics.f1_score(ytrain,trainp))

print("F1 score of test",metrics.f1_score(ytest,testp))
```

Naive Bayes

In [ ]:

```
model=naive_bayes.GaussianNB()

model.fit(Xtrain[cols],ytrain)

trainp=model.predict(Xtrain[cols])
testp=model.predict(Xtest[cols])

print("Accuracy of training",metrics.accuracy_score(ytrain,trainp))

print("Accuracy of test",metrics.accuracy_score(ytest,testp))

print("Precision of training",metrics.precision_score(ytrain,trainp))

print("Precision of test",metrics.precision_score(ytest,testp))

print("Confusion matrix of training",metrics.confusion_matrix(ytrain,trainp))

print("Confusion matrix of test",metrics.confusion_matrix(ytest,testp))

print("AUC of training",metrics.roc_auc_score(ytrain,trainp))

print("AUC of test",metrics.roc_auc_score(ytest,testp))

print("F1 score of training",metrics.f1_score(ytrain,trainp))

print("F1 score of test",metrics.f1_score(ytest,testp))
```

Random Forest

In [ ]:

```
model=ensemble.RandomForestClassifier()
```

```
model.fit(Xtrain[cols],ytrain)

trainp=model.predict(Xtrain[cols])
testp=model.predict(Xtest[cols])

print("Accuracy of training",metrics.accuracy_score(ytrain,trainp))

print("Accuracy of test",metrics.accuracy_score(ytest,testp))

print("Precision of training",metrics.precision_score(ytrain,trainp))

print("Precision of test",metrics.precision_score(ytest,testp))

print("Confusion matrix of training",metrics.confusion_matrix(ytrain,trainp))

print("Confusion matrix of test",metrics.confusion_matrix(ytest,testp))

print("AUC of training",metrics.roc_auc_score(ytrain,trainp))

print("AUC of test",metrics.roc_auc_score(ytest,testp))

print("F1 score of training",metrics.f1_score(ytrain,trainp))

print("F1 score of test",metrics.f1_score(ytest,testp))
```

All model in a function

In [ ]:

```
from sklearn import tree
from sklearn import neighbors
from sklearn import naive_bayes
from sklearn import ensemble



def modelstats1(Xtrain,Xtest,ytrain,ytest):
    stats=[]
    modelnames=["LR","DecisionTree","NB","RF"]#,"KNN"
    models=list()
    models.append(linear_model.LogisticRegression())

models.append(tree.DecisionTreeClassifier(max_depth=10,min_samples_split=27,min_samples
_leaf=22,random_state=100))
    #models.append(neighbors.KNeighborsClassifier())
    models.append(naive_bayes.GaussianNB())
    models.append(ensemble.RandomForestClassifier())
    for name,model in zip(modelnames,models):
        if name=="KNN":
            k=[l for l in range(5,17,2)]
            grid={"n_neighbors":k}

grid_obj=model_selection.GridSearchCV(estimator=model,param_grid=grid,scoring="f1")
            grid_fit=grid_obj.fit(Xtrain,ytrain)
            model =grid_fit.best_estimator_
            model.fit(Xtrain,ytrain)
            name=name+"("+str(grid_fit.best_params_["n_neighbors"])+")"
```

```
        else:
            model.fit(Xtrain,ytrain)
            trainprediction=model.predict(Xtrain)
            testprediction=model.predict(Xtest)
        scores=list()
        scores.append(name+"-train")
        scores.append(metrics.accuracy_score(ytrain,trainprediction))
        scores.append(metrics.precision_score(ytrain,trainprediction))
        scores.append(metrics.recall_score(ytrain,trainprediction))
        scores.append(metrics.roc_auc_score(ytrain,trainprediction))
        scores.append(metrics.f1_score(ytrain,trainprediction))
        stats.append(scores)
        scores=list()
        scores.append(name+"-test")
        scores.append(metrics.accuracy_score(ytest,testprediction))
        scores.append(metrics.precision_score(ytest,testprediction))
        scores.append(metrics.recall_score(ytest,testprediction))
        scores.append(metrics.roc_auc_score(ytest,testprediction))
        scores.append(metrics.f1_score(ytest,testprediction))
        stats.append(scores)
    colnames=["MODELNAME","ACCURACY","PRECISION","RECALL","AUC","F1_SCORE"]
    return pd.DataFrame(stats,columns=colnames)
```

In [ ]:

```
modelstats1(Xtrain[cols],Xtest[cols],ytrain,ytest)
```

Future Scope of Improvements

Our given data set contains very large amount pf null value in it. And the given values of columns are having very large number of outliers but as the values are normally distributed we can't remove outliers for building model. If the given data set contains less null values then the output of our project would be better.

So if we get some good data set with less outliers then there will be a huge difference in scores of our project.

The result could be better if we used Neural Network in our project.

Reference

We have taken reference from these website:

https://www.kaggle.com/

https://data-flair.training/blogs/machine-learning-project-ideas/

https://scikit-learn.org/stable/

https://pandas.pydata.org/

https://numpy.org/

https://scikit-learn.org/stable/modules/generated/sklearn.feature_selection.RFE.html

https://towardsdatascience.com/model-or-do-you-mean-weight-of-evidence-woe-and-information-value-iv-331499f6fc2

https://serokell.io/blog/machine-learning-project-ideas

Certificate

This is to certify that Mr Riju Chatterjee of Kalyani Government Engineering College, registration number: 009778 of 2019-2020, has successfully completed a project on Claims Management of Machine Learning using Python using Anaconda Jupyter Notebook under the guidance of Mr Titas RoyChowdhury.

Titas RoyChowdhury.

Globsyn Finishing School

Certificate

This is to certify that Mr Ayan Ghorai of Kalyani Government Engineering College, registration number: 181020110016 of 2018-2019, has successfully completed a project on Claims Management of Machine Learning using Python using Anaconda Jupyter Notebook under the guidance of Mr Titas RoyChowdhury.

Titas RoyChowdhury.

Globsyn Finishing School

Certificate

This is to certify that Mr Debjit Adhikary of Kalyani Government Engineering College, registration number: 181020110063 of 2018-2019, has successfully completed a project on Claims Management of Machine Learning using Python using Anaconda Jupyter Notebook under the guidance of Mr Titas RoyChowdhury.

Titas RoyChowdhury.

Globsyn Finishing School

Certificate

This is to certify that Mr Subhodip Ghosh of Kalyani Government Engineering College, registration number: 181020110045 of 2018-2019, has successfully completed a project on Claims Management of Machine Learning using Python using Anaconda Jupyter Notebook under the guidance of Mr Titas RoyChowdhury.

Titas RoyChowdhury.

Globsyn Finishing School

Certificate

This is to certify that Ms Shreya Mallick of Kalyani Government Engineering College, registration number: 201020101610006 of 2020-21, has successfully completed a project on Claims Management of Machine Learning using Python using Anaconda Jupyter Notebook under the guidance of Mr Titas RoyChowdhury.

Titas RoyChowdhury.

Globsyn Finishing School