ReneWind Project

By: Esra Mercan

Contents

- Business Problem Overview
- Data Overview
- Exploratory Data Analysis (EDA)
- Data pre-processing
- Model building
- Model building - Oversampled data
- Model building - Undersampled data
- Hyperparameter tuning
- Productionize the model
- Model Performance Summary
- Business Insights and Recommendations

Context

Renewable energy sources play an increasingly important role in the global energy mix, as the effort to reduce the environmental impact of energy production increases.

Out of all the renewable energy alternatives, wind energy is one of the most developed technologies worldwide. The U.S Department of Energy has put together a guide to achieving operational efficiency using predictive maintenance practices.

Predictive maintenance uses sensor information and analysis methods to measure and predict degradation and future component capability. The idea behind predictive maintenance is that failure patterns are predictable and if component failure can be predicted accurately and the component is replaced before it fails, the costs of operation and maintenance will be much lower.

The sensors fitted across different machines involved in the process of energy generation collect data related to various environmental factors (temperature, humidity, wind speed, etc.) and additional features related to various parts of the wind turbine (gearbox, tower, blades, break, etc.).

Objective

“ReneWind” is a company working on improving the machinery/processes involved in the production of wind energy using machine learning and has collected data of generator failure of wind turbines using sensors. They have shared a ciphered version of the data, as the data collected through sensors is confidential (the type of data collected varies with companies). Data has 40 predictors, 20000 observations in the training set and 5000 in the test set.

The objective is to build various classification models, tune them, and find the best one that will help identify failures so that the generators could be repaired before failing/breaking to reduce the overall maintenance cost. The nature of predictions made by the classification model will translate as follows:

- True positives (TP) are failures correctly predicted by the model. These will result in repairing costs.
- False negatives (FN) are real failures where there is no detection by the model. These will result in replacement costs.
- False positives (FP) are detections where there is no failure. These will result in inspection costs.

It is given that the cost of repairing a generator is much less than the cost of replacing it, and the cost of inspection is less than the cost of repair.

“1” in the target variables should be considered as “failure” and “0” represents “No failure”

Data Overview

- The data provided is a transformed version of original data which was collected using sensors.
- Train.csv - To be used for training and tuning of models.
- Test.csv - To be used only for testing the performance of the final best model.
- Both the datasets consist of 40 predictor variables and 1 target variable

Exploratory Data Analysis (EDA)

This is how our data looks in the beginning:

	V1	V2	V3	V4	V5	V6	V7	V8	V9	V10	V11	V12	V13	V14	V15	V16	V17	V18	V19	V20
0	-4.465	-4.679	3.102	0.506	-0.221	-2.033	-2.911	0.051	-1.522	3.762	-5.715	0.736	0.981	1.418	-3.376	-3.047	0.306	2.914	2.270	4.395
1	3.366	3.653	0.910	-1.368	0.332	2.359	0.733	-4.332	0.566	-0.101	1.914	-0.951	-1.255	-2.707	0.193	-4.769	-2.205	0.908	0.757	-5.834
2	-3.832	-5.824	0.634	-2.419	-1.774	1.017	-2.099	-3.173	-2.082	5.393	-0.771	1.107	1.144	0.943	-3.164	-4.248	-4.039	3.689	3.311	1.059
3	1.618	1.888	7.046	-1.147	0.083	-1.530	0.207	-2.494	0.345	2.119	-3.053	0.460	2.705	-0.636	-0.454	-3.174	-3.404	-1.282	1.582	-1.952
4	-0.111	3.872	-3.758	-2.983	3.793	0.545	0.205	4.849	-1.855	-6.220	1.998	4.724	0.709	-1.989	-2.633	4.184	2.245	3.734	-6.313	-5.380

	V21	V22	V23	V24	V25	V26	V27	V28	V29	V30	V31	V32	V33	V34	V35	V36	V37	V38	V39	V40	Target
0	-2.388	0.646	-1.191	3.133	0.665	-2.511	-0.037	0.726	-3.982	-1.073	1.667	3.060	-1.690	2.846	2.235	6.667	0.444	-2.369	2.951	-3.480	0
1	-3.065	1.597	-1.757	1.766	-0.267	3.625	1.500	-0.586	0.783	-0.201	0.025	-1.795	3.033	-2.468	1.895	-2.298	-1.731	5.909	-0.386	0.616	0
2	-2.143	1.650	-1.661	1.680	-0.451	-4.551	3.739	1.134	-2.034	0.841	-1.600	-0.257	0.804	4.086	2.292	5.361	0.352	2.940	3.839	-4.309	0
3	-3.517	-1.206	-5.628	-1.818	2.124	5.295	4.748	-2.309	-3.963	-6.029	4.949	-3.584	-2.577	1.364	0.623	5.550	-1.527	0.139	3.101	-1.277	0
4	-0.887	2.062	9.446	4.490	-3.945	4.582	-8.780	-3.383	5.107	6.788	2.044	8.266	6.629	-10.069	1.223	-3.230	1.687	-2.164	-3.645	6.510	0

This is how our data looks and in the end

	V1	V2	V3	V4	V5	V6	V7	V8	V9	V10	V11	V12	V13	V14	V15	V16	V17	V18	V19	V20
19995	-2.071	-1.088	-0.796	-3.012	-2.288	2.807	0.481	0.105	-0.587	-2.899	8.868	1.717	1.358	-1.777	0.710	4.945	-3.100	-1.199	-1.085	-0.365
19996	2.890	2.483	5.644	0.937	-1.381	0.412	-1.593	-5.762	2.150	0.272	-2.095	-1.526	0.072	-3.540	-2.762	-10.632	-0.495	1.720	3.872	-1.210
19997	-3.897	-3.942	-0.351	-2.417	1.108	-1.528	-3.520	2.055	-0.234	-0.358	-3.782	2.180	6.112	1.985	-8.330	-1.639	-0.915	5.672	-3.924	2.133
19998	-3.187	-10.052	5.696	-4.370	-5.355	-1.873	-3.947	0.679	-2.389	5.457	1.583	3.571	9.227	2.554	-7.039	-0.994	-9.665	1.155	3.877	3.524
19999	-2.687	1.961	6.137	2.600	2.657	-4.291	-2.344	0.974	-1.027	0.497	-9.589	3.177	1.055	-1.416	-4.669	-5.405	3.720	2.893	2.329	1.458

	V21	V22	V23	V24	V25	V26	V27	V28	V29	V30	V31	V32	V33	V34	V35	V36	V37	V38	V39	V40	Target
19995	3.131	-3.948	-3.578	-8.139	-1.937	-1.328	-0.403	-1.735	9.996	6.955	-3.938	-8.274	5.745	0.589	-0.650	-3.043	2.216	0.609	0.178	2.928	1
19996	-8.222	2.121	-5.492	1.452	1.450	3.685	1.077	-0.384	-0.839	-0.748	-1.089	-4.159	1.181	-0.742	5.369	-0.693	-1.669	3.660	0.820	-1.987	0
19997	-4.502	2.777	5.728	1.620	-1.700	-0.042	-2.923	-2.760	-2.254	2.552	0.982	7.112	1.476	-3.954	1.856	5.029	2.083	-6.409	1.477	-0.874	0
19998	-7.015	-0.132	-3.446	-4.801	-0.876	-3.812	5.422	-3.732	0.609	5.256	1.915	0.403	3.164	3.752	8.530	8.451	0.204	-7.130	4.249	-6.112	0
19999	-6.429	1.818	0.806	7.786	0.331	5.257	-4.867	-0.819	-5.667	-2.861	4.674	6.621	-1.989	-1.349	3.952	5.450	-0.455	-2.202	1.678	-1.974	0

```
df.shape ##
```

```
(20000, 41)
```

There are 20000 rows and 41 column in the training data set

```
df_test.shape
```

```
(5000, 41)
```

There are 5000 rows and 41 column in the testing data set.

```
# checking for duplicate
data.duplicated().sum()
```

```
0
```

There is no duplicate values.

Exploratory Data Analysis (EDA)

Data Info

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 20000 entries, 0 to 19999
Data columns (total 41 columns):
 #   Column   Non-Null Count  Dtype
---  ------   --------------  -----
 0   V1       19982 non-null  float64
 1   V2       19982 non-null  float64
 2   V3       20000 non-null  float64
 3   V4       20000 non-null  float64
 4   V5       20000 non-null  float64
 5   V6       20000 non-null  float64
 6   V7       20000 non-null  float64
 7   V8       20000 non-null  float64
 8   V9       20000 non-null  float64
 9   V10      20000 non-null  float64
 10  V11      20000 non-null  float64
 11  V12      20000 non-null  float64
 12  V13      20000 non-null  float64
 13  V14      20000 non-null  float64
 14  V15      20000 non-null  float64
 15  V16      20000 non-null  float64
 16  V17      20000 non-null  float64
 17  V18      20000 non-null  float64
 18  V19      20000 non-null  float64
 19  V20      20000 non-null  float64
 20  V21      20000 non-null  float64
 21  V22      20000 non-null  float64
 22  V23      20000 non-null  float64
 23  V24      20000 non-null  float64
 24  V25      20000 non-null  float64
 25  V26      20000 non-null  float64
 26  V27      20000 non-null  float64
 27  V28      20000 non-null  float64
 28  V29      20000 non-null  float64
 29  V30      20000 non-null  float64
 30  V31      20000 non-null  float64
 31  V32      20000 non-null  float64
 32  V33      20000 non-null  float64
 33  V34      20000 non-null  float64
 34  V35      20000 non-null  float64
 35  V36      20000 non-null  float64
 36  V37      20000 non-null  float64
 37  V38      20000 non-null  float64
 38  V39      20000 non-null  float64
 39  V40      20000 non-null  float64
 40  Target   20000 non-null  int64
dtypes: float64(40), int64(1)
memory usage: 6.3 MB
```

We have all numerical variables in the dataset. V1 and V2 have 18 missing values

Let's check the statistical summary of the data.

	count	mean	std	min	25%	50%	75%	max
V1	19982.000	-0.272	3.442	-11.876	-2.737	-0.748	1.840	15.493
V2	19982.000	0.440	3.151	-12.320	-1.641	0.472	2.544	13.089
V3	20000.000	2.485	3.389	-10.708	0.207	2.256	4.566	17.091
V4	20000.000	-0.083	3.432	-15.082	-2.348	-0.135	2.131	13.236
V5	20000.000	-0.054	2.105	-8.603	-1.536	-0.102	1.340	8.134
V6	20000.000	-0.995	2.041	-10.227	-2.347	-1.001	0.380	6.976
V7	20000.000	-0.879	1.762	-7.950	-2.031	-0.917	0.224	8.006
V8	20000.000	-0.548	3.296	-15.658	-2.643	-0.389	1.723	11.679
V9	20000.000	-0.017	2.161	-8.596	-1.495	-0.068	1.409	8.138
V10	20000.000	-0.013	2.193	-9.854	-1.411	0.101	1.477	8.108
V11	20000.000	-1.895	3.124	-14.832	-3.922	-1.921	0.119	11.826
V12	20000.000	1.605	2.930	-12.948	-0.397	1.508	3.571	15.081
V13	20000.000	1.580	2.875	-13.228	-0.224	1.637	3.460	15.420
V14	20000.000	-0.951	1.790	-7.739	-2.171	-0.957	0.271	5.671
V15	20000.000	-2.415	3.355	-16.417	-4.415	-2.383	-0.359	12.246
V16	20000.000	-2.925	4.222	-20.374	-5.634	-2.683	-0.095	13.583
V17	20000.000	-0.134	3.345	-14.091	-2.216	-0.015	2.069	16.756
V18	20000.000	1.189	2.592	-11.644	-0.404	0.883	2.572	13.180
V19	20000.000	1.182	3.397	-13.492	-1.050	1.279	3.493	13.238
V20	20000.000	0.024	3.669	-13.923	-2.433	0.033	2.512	16.052
V21	20000.000	-3.611	3.568	-17.956	-5.930	-3.533	-1.266	13.840
V22	20000.000	0.952	1.652	-10.122	-0.118	0.975	2.026	7.410
V23	20000.000	-0.366	4.032	-14.866	-3.099	-0.262	2.452	14.459
V24	20000.000	1.134	3.912	-16.387	-1.468	0.969	3.546	17.163
V25	20000.000	-0.002	2.017	-8.228	-1.365	0.025	1.397	8.223
V26	20000.000	1.874	3.435	-11.834	-0.338	1.951	4.130	16.836
V27	20000.000	-0.612	4.369	-14.905	-3.652	-0.885	2.189	17.560
V28	20000.000	-0.883	1.918	-9.269	-2.171	-0.891	0.376	6.528
V29	20000.000	-0.986	2.684	-12.579	-2.787	-1.176	0.630	10.722
V30	20000.000	-0.016	3.005	-14.796	-1.867	0.184	2.036	12.506
V31	20000.000	0.487	3.461	-13.723	-1.818	0.490	2.731	17.255
V32	20000.000	0.304	5.500	-19.877	-3.420	0.052	3.762	23.633
V33	20000.000	0.050	3.575	-16.898	-2.243	-0.066	2.255	16.692
V34	20000.000	-0.463	3.184	-17.985	-2.137	-0.255	1.437	14.358
V35	20000.000	2.230	2.937	-15.350	0.336	2.099	4.064	15.291
V36	20000.000	1.515	3.801	-14.833	-0.944	1.567	3.984	19.330
V37	20000.000	0.011	1.788	-5.478	-1.256	-0.128	1.176	7.467
V38	20000.000	-0.344	3.948	-17.375	-2.988	-0.317	2.279	15.290
V39	20000.000	0.891	1.753	-6.439	-0.272	0.919	2.058	7.760
V40	20000.000	-0.876	3.012	-11.024	-2.940	-0.921	1.120	10.654
Target	20000.000	0.056	0.229	0.000	0.000	0.000	0.000	1.000

V16, V23,V27 and V32 have slighly high std comparing to other variables. Almost all variables mean and median is pretty close to each other. We should she a normal distribution except the target variable.

Missing entries in the train data

```
V1        18
V2        18
V3         0
V4         0
V5         0
V6         0
V7         0
V8         0
V9         0
V10        0
V11        0
V12        0
V13        0
V14        0
V15        0
V16        0
V17        0
V18        0
V19        0
V20        0
V21        0
V22        0
V23        0
V24        0
V25        0
V26        0
V27        0
V28        0
V29        0
V30        0
V31        0
V32        0
V33        0
V34        0
V35        0
V36        0
V37        0
V38        0
V39        0
V40        0
Target     0
dtype: int64
```

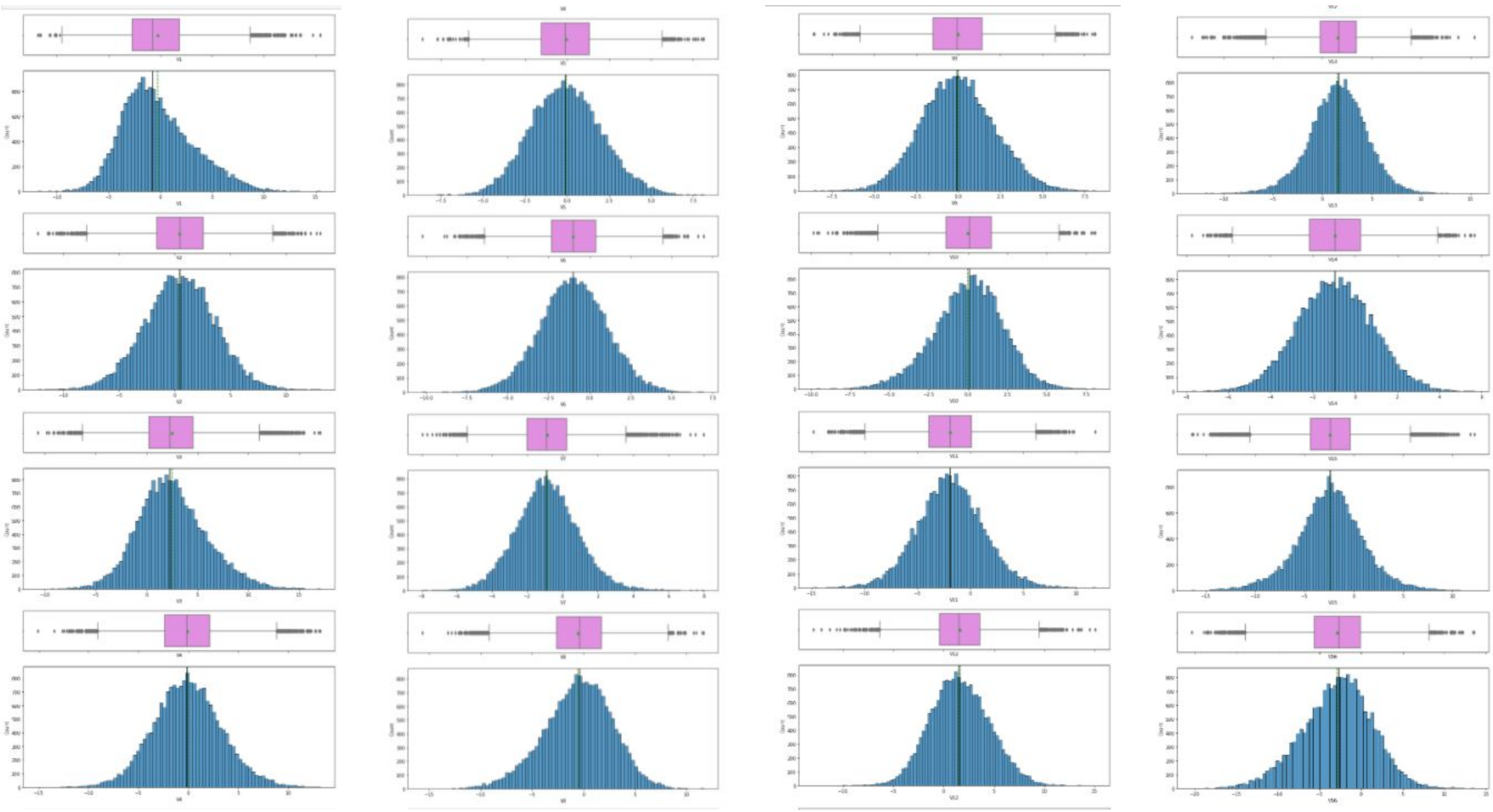
V1 and V2 have 18 missing values

Missing entries in the test data

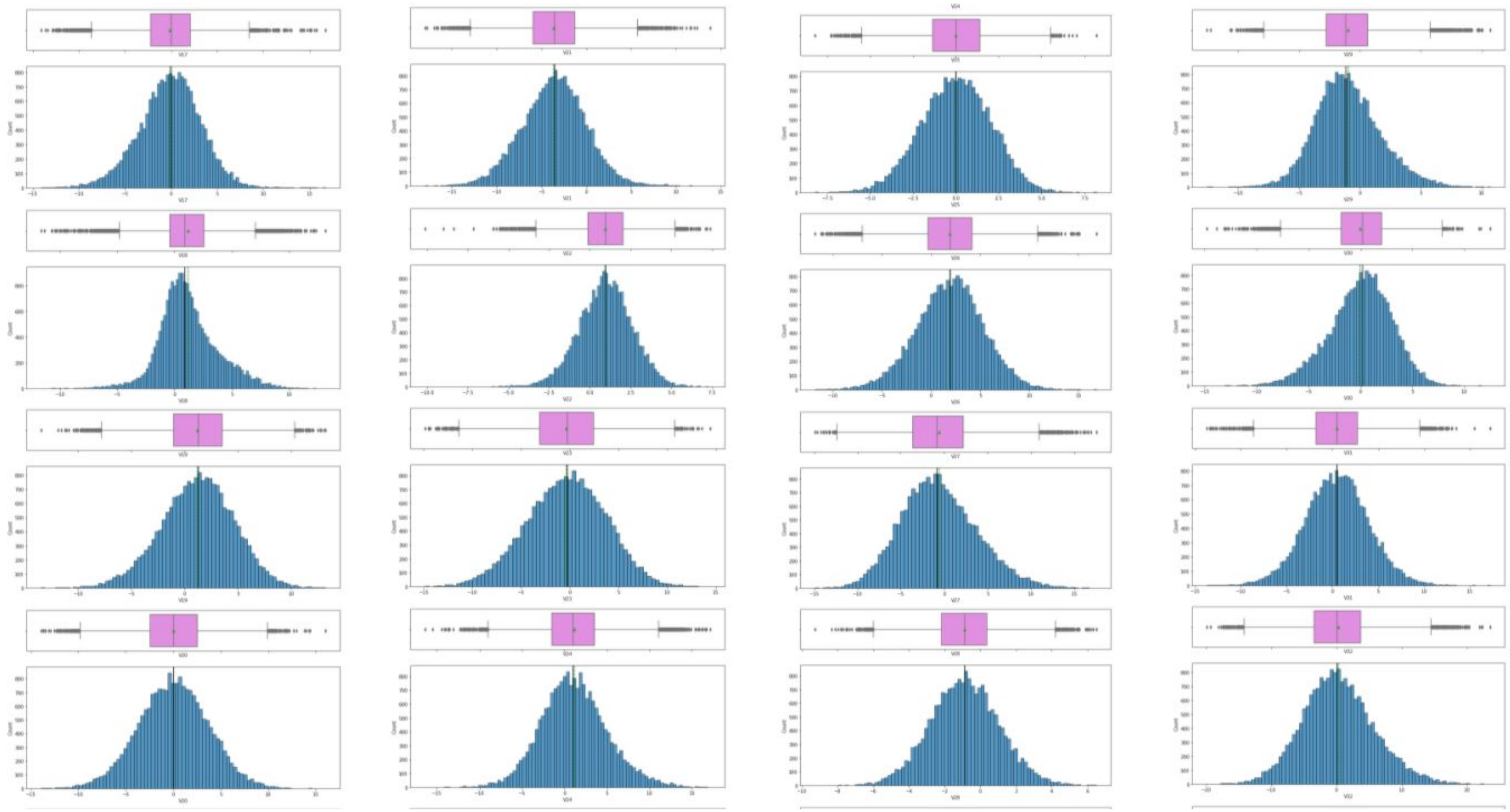
```
V1        5
V2        6
V3        0
V4        0
V5        0
V6        0
V7        0
V8        0
V9        0
V10       0
V11       0
V12       0
V13       0
V14       0
V15       0
V16       0
V17       0
V18       0
V19       0
V20       0
V21       0
V22       0
V23       0
V24       0
V25       0
V26       0
V27       0
V28       0
V29       0
V30       0
V31       0
V32       0
V33       0
V34       0
V35       0
V36       0
V37       0
V38       0
V39       0
V40       0
Target    0
dtype: int64
```

V1 has 5 and V2 has 6 missing values

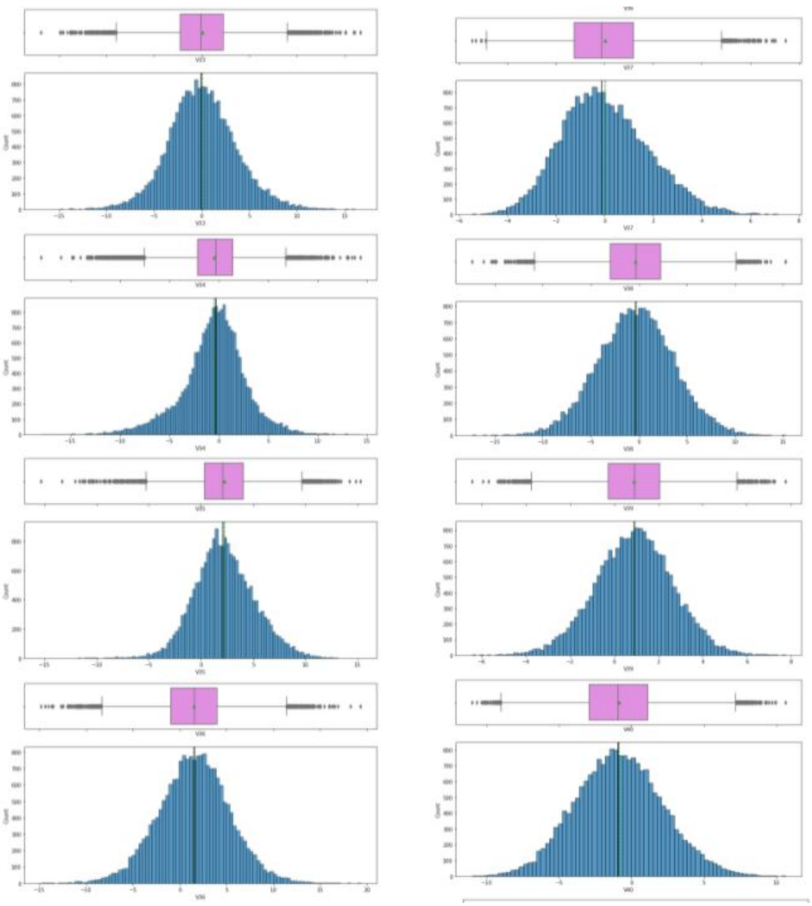
Exploratory Data Analysis (EDA)

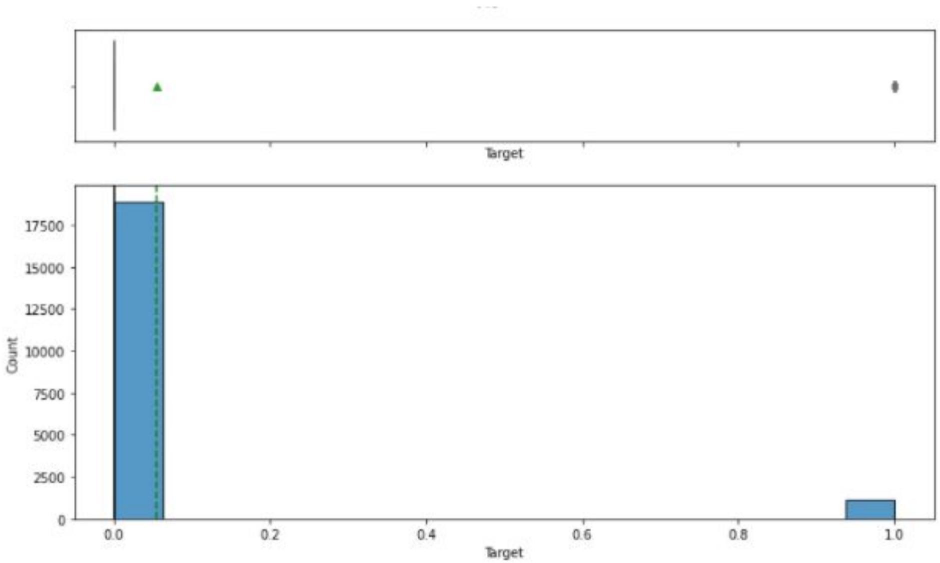


Exploratory Data Analysis (EDA) - continued



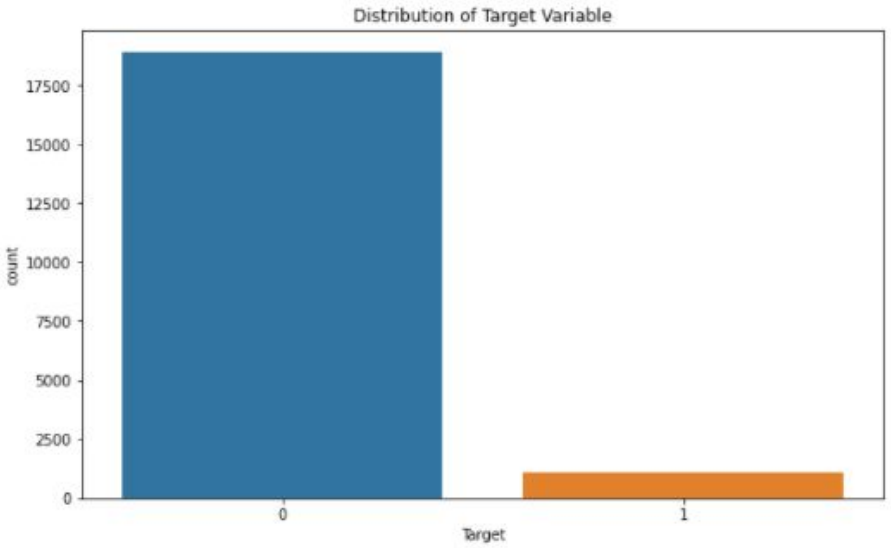
Exploratory Data Analysis (EDA) - continued



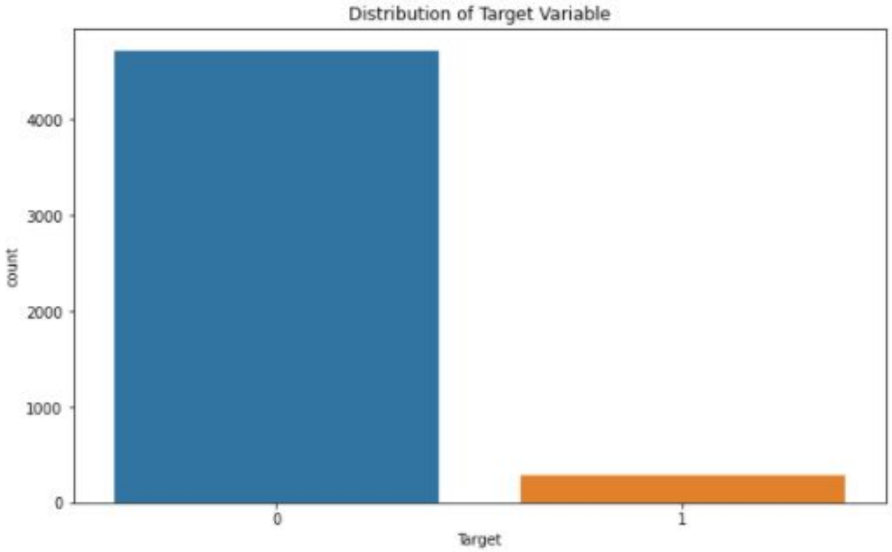


Almost all variables has normal distribution. Majority of the target variable has 0 . There some skewed variables but we will not treat them.

Exploratory Data Analysis (EDA)-Univariate Analysis



As we see above majority of the training data has 0 which means No failure.



Majority of the test data has 0, which means No failure on most of the variables.

Data pre-processing

Splitting data into training, validation and test set:

```
(16000, 40) (4000, 40) (5000, 40)
```

Checking that no column has missing values in train or test sets

```
V1     0
V2     0
V3     0
V4     0
V5     0
V6     0
V7     0
V8     0
V9     0
V10    0
V11    0
V12    0
V13    0
V14    0
V15    0
V16    0
V17    0
V18    0
V19    0
V20    0
V21    0
V22    0
V23    0
V24    0
V25    0
V26    0
V27    0
V28    0
V29    0
V30    0
V31    0
V32    0
V33    0
V34    0
V35    0
V36    0
V37    0
V38    0
V39    0
V40    0
dtype: int64
```

- We used SimpleImputer mode to impute missing values in the columns
- All missing values have been treated.

Model building

Model evaluation criterion

The nature of predictions made by the classification model will translate as follows:

- True positives (TP) are failures correctly predicted by the model.
- False negatives (FN) are real failures in a generator where there is no detection by model.
- False positives (FP) are failure detections in a generator where there is no failure.

Which metric to optimize?

- We need to choose the metric which will ensure that the maximum number of generator failures are predicted correctly by the model.
- We would want Recall to be maximized as greater the Recall, the higher the chances of minimizing false negatives.
- We want to minimize false negatives because if a model predicts that a machine will have no failure when there will be a failure, it will increase the maintenance cost.

Defining scorer to be used for cross-validation and hyperparameter tuning

- We want to reduce false negatives and will try to maximize "Recall".
- To maximize Recall, we can use Recall as a **scorer** in cross-validation and hyperparameter tuning.

Model Building on original data

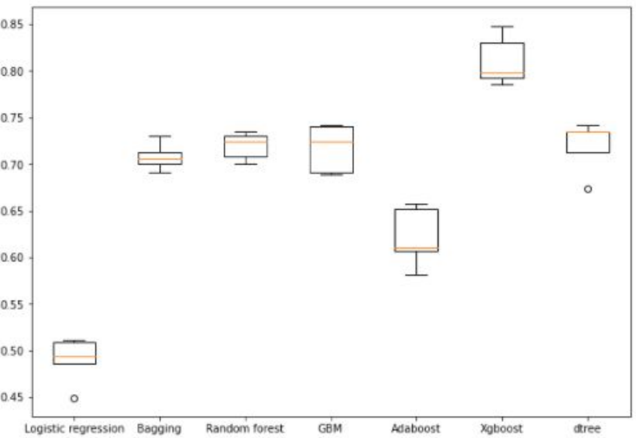
```
Cross-Validation Cost:

Logistic regression: 0.48988129245223133
Bagging: 0.7083222243382213
Random forest: 0.7195899193804354
GBM: 0.7173363803719928
Adaboost: 0.6215641465117756
Xgboost: 0.810804291246112
dtree: 0.7196280073636767

Validation Performance:

Logistic regression: 0.49099099099099097
Bagging: 0.7207207207207207
Random forest: 0.7432432432432432
GBM: 0.7432432432432432
Adaboost: 0.6576576576576577
Xgboost: 0.8153153153153153
dtree: 0.7387387387387387
```

Algorithm Comparison



Model Building with undersampled data

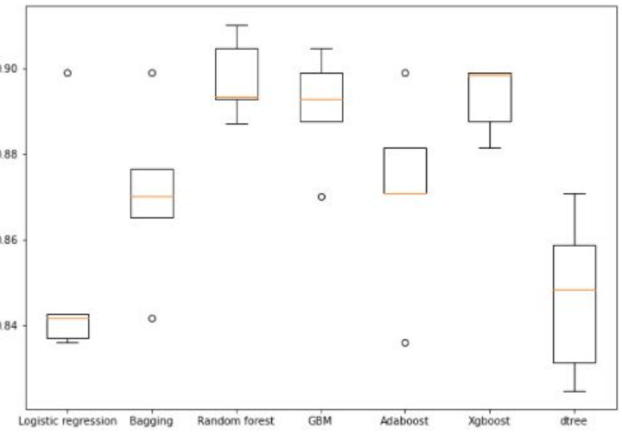
```
Cross-Validation Cost:

Logistic regression: 0.8513235574176348
Bagging: 0.8704627689963816
Random forest: 0.8975052370976957
GBM: 0.8907446200723672
Adaboost: 0.8715927124992063
Xgboost: 0.8930108550752237
dtree: 0.8468355233923697

Validation Performance:

Logistic regression: 0.8648648648648649
Bagging: 0.8918918918918919
Random forest: 0.8783783783783784
GBM: 0.8873873873873874
Adaboost: 0.8558558558558559
Xgboost: 0.8918918918918919
dtree: 0.8468468468468469
```

undersampled data



Model Building with oversampled data

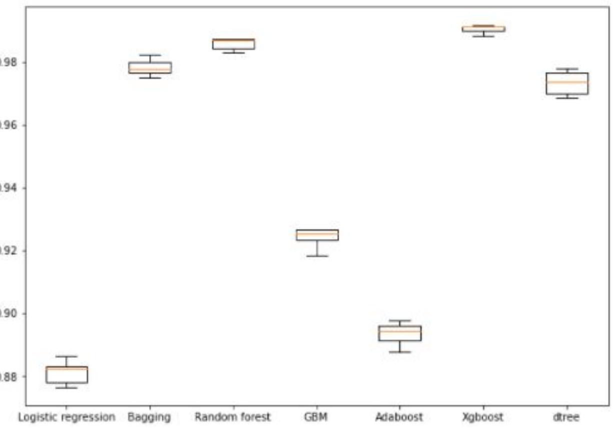
```
Cross-Validation Cost:

Logistic regression: 0.8812865538044636
Bagging: 0.9781630048735123
Random forest: 0.9855744607906776
GBM: 0.9239674518302545
Adaboost: 0.8935280870047044
Xgboost: 0.9906035856141958
dtree: 0.9732668119313808

Validation Performance:

Logistic regression: 0.8513513513513513
Bagging: 0.8423423423423423
Random forest: 0.8558558558558559
GBM: 0.8828828828828829
Adaboost: 0.8558558558558559
Xgboost: 0.8693693693693694
dtree: 0.8198198198198198
```

oversampled data



- We can see that the xgboost is giving the highest cross-validated recall followed by GBM, Adaboost and Random Forest
- The boxplot shows that the performance of xgboost, GBM, Adaboost and Random Forest is consistent and their performance on the validation set is also good
- We will tune the best 4 models i.e. and see if the performance improves

Hyperparameter Tuning

Tuning AdaBoost using oversampled data

Best parameters are {'n_estimators': 200, 'learning_rate': 0.2, 'base_estimator': DecisionTreeClassifier(max_depth=3, random_state=1)} with CV score=0.9693618503452355:
Wall time: 12min 18s

Creating new pipeline with best parameters

AdaBoostClassifier(base_estimator=DecisionTreeClassifier(max_depth=3,

random_state=1),
learning_rate=0.2,
n_estimators=200)performance on

performance on oversampled train set

	Accuracy	Recall	Precision	F1
0	0.989	0.984	0.995	0.989

the performance on validation set

	Accuracy	Recall	Precision	F1
0	0.982	0.860	0.820	0.840

Tuning Random forest using undersampled data

Best parameters are {'n_estimators': 300, 'min_samples_leaf': 1, 'max_samples': 0.4, 'max_features': 'sqrt'} with CV score=0.8953278740557353:
Wall time: 21.2 s

Creating new pipeline with best parameters

RandomForestClassifier(max_features='sqrt', max_samples=0.4, n_estimators=300,
random_state=1)

performance on undersampled train set

	Accuracy	Recall	Precision	F1
0	0.967	0.940	0.993	0.966

the performance on validation set

	Accuracy	Recall	Precision	F1
0	0.933	0.878	0.449	0.595

Hyperparameter Tuning-continued

Tuning Gradient Boosting using oversampled data

Best parameters are {'subsample': 0.7, 'n_estimators': 125, 'max_features': 0.7, 'learning_rate': 1} with CV score=0.9708836489188448:
Wall time: 5min 9s

Creating new pipeline with best parameters

```
GradientBoostingClassifier(learning_rate=1,
max_features=0.7, n_estimators=125,
                      random_state=1,
subsample=0.7)
```

performance on oversampled train set

	Accuracy	Recall	Precision	F1
0	0.994	0.994	0.994	0.994

the performance on validation set

	Accuracy	Recall	Precision	F1
0	0.967	0.860	0.656	0.745

Tuning XGBoost using oversampled data

Best parameters are {'subsample': 0.8, 'scale_pos_weight': 10, 'n_estimators': 250, 'learning_rate': 0.1, 'gamma': 5} with CV score=0.9960296014254713:
Wall time: 39min 51s

fitting the model on over sampled data

```
XGBClassifier(base_score=0.5, booster='gbtree', colsample_bylevel=1,
      colsample_bynode=1, colsample_bytree=1, enable_categorical=False,
      eval_metric='logloss', gamma=5, gpu_id=-1, importance_type=None,
      interaction_constraints='', learning_rate=0.1, max_delta_step=0,
      max_depth=6, min_child_weight=1, missing=nan,
      monotone_constraints='()', n_estimators=250, n_jobs=8,
      num_parallel_tree=1, predictor='auto', random_state=1,
      reg_alpha=0, reg_lambda=1, scale_pos_weight=10, subsample=0.8,
      tree_method='exact', validate_parameters=1, verbosity=None)
```

performance on oversampled train set

	Accuracy	Recall	Precision	F1
0	0.998	1.000	0.996	0.998

the performance on validation set

	Accuracy	Recall	Precision	F1
0	0.979	0.874	0.773	0.820

Model Performance comparison

Training performance comparison after Hyperparameter Tuning:

	Gradient Boosting tuned with oversampled data	AdaBoost classifier tuned with oversampled data	Random forest tuned with undersampled data	XGBoost tuned with oversampled data
Accuracy	0.967	0.989	0.967	0.998
Recall	0.860	0.984	0.940	1.000
Precision	0.656	0.995	0.993	0.996
F1	0.745	0.989	0.966	0.998

Validation performance comparison after Hyperparameter Tuning:

	Gradient Boosting tuned with oversampled data	AdaBoost classifier tuned with oversampled data	Random forest tuned with undersampled data	XGBoost tuned with oversampled data
Accuracy	0.967	0.982	0.933	0.979
Recall	0.860	0.860	0.878	0.874
Precision	0.656	0.820	0.449	0.773
F1	0.745	0.840	0.595	0.820

Let's check the performance on unseen test data before the pipeline:

	Accuracy	Recall	Precision	F1
0	0.821	0.826	0.216	0.342

XGboost has the best performance score on original, undersampled and oversampled data. I will choose that for my model.

Pipeline Application

```
pipe = Pipeline([('imputer', SimpleImputer(strategy='median')), ("XGB",
            XGBClassifier(base_score=0.5,
                                   colsample_bylevel=1, colsample_bynode=1,
                                   colsample_bytree=1, enable_categorical=False,
                                   eval_metric='logloss', gamma=0, gpu_id=-1,
                                   importance_type=None, interaction_constraints='',
                                   learning_rate=0.1, max_delta_step=0, max_depth=6,
                                   min_child_weight=1,
                                   monotone_constraints='()', n_estimators=250,
                                   n_jobs=8, num_parallel_tree=1, predictor='auto',
                                   random_state=1, reg_alpha=0, reg_lambda=1,
                                   scale_pos_weight=10, subsample=0.8,
                                   tree_method='exact', validate_parameters=1,
                                   verbosity=None,
            ),
        ),
    ]
)
```

I used xgboost for my pipeline.

Best Model Performance

	Accuracy	Recall	Precision	F1
0	0.977	0.858	0.761	0.807

```
# Separating target variable and other variables
X1 = data.drop(columns="Target", axis=1)
Y1 = data["Target"]

# Since we already have a separate test set, we don't need to divide da

X_test1 = df_test.drop(columns="Target" , axis=1) ## Complete the code
y_test1 = df_test["Target"] ## Complete the code to store target vari
```

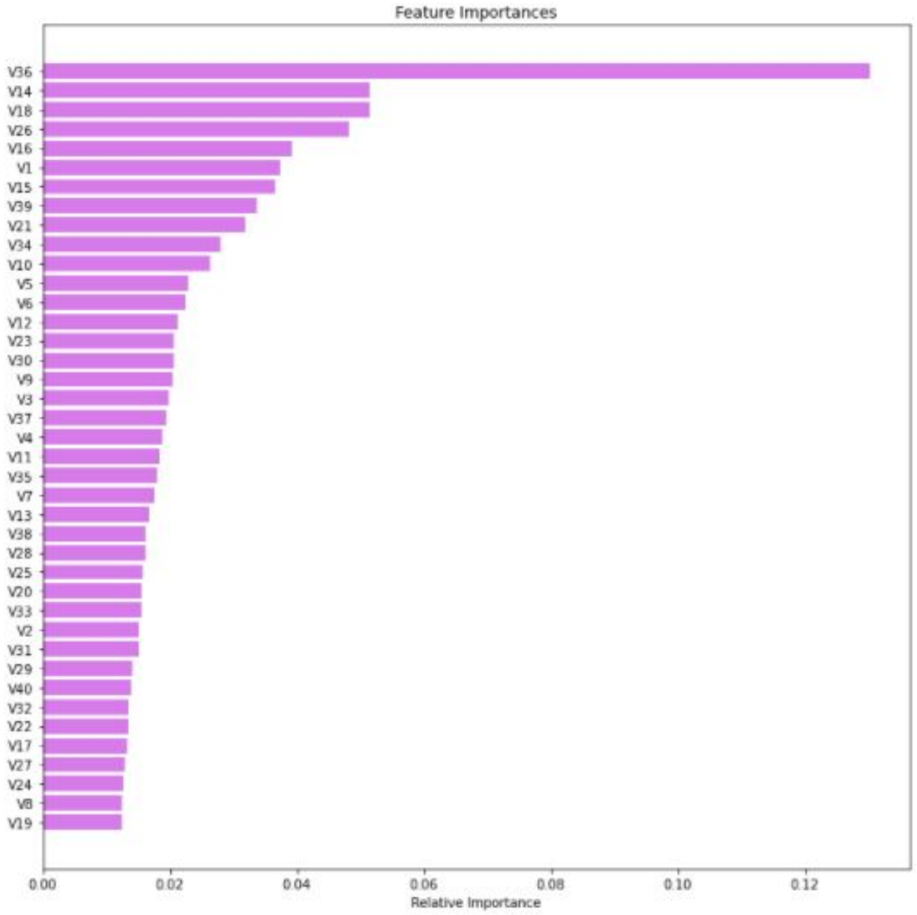
```
imputer = SimpleImputer(strategy="median")
X1 = imputer.fit_transform(X1)
```

```
# Synthetic Minority Over Sampling Technique
sm = SMOTE(sampling_strategy=1, k_neighbors=5, random_state=1)
X_over1, y_over1 = sm.fit_resample(X1, Y1)
```

```
pipe.fit(X_over1, y_over1) ## Complete the code to fit the Model obtained from above step
```

```
Pipeline(steps=[('imputer', SimpleImputer(strategy='median')),
                ('XGB',
                 XGBClassifier(base_score=0.5, booster='gbtree',
                               colsample_bylevel=1, colsample_bynode=1,
                               colsample_bytree=1, enable_categorical=False,
                               eval_metric='logloss', gamma=0, gpu_id=-1,
                               importance_type=None, interaction_constraints='',
                               learning_rate=0.1, max_delta_step=0, max_depth=6,
                               min_child_weight=1, missing=nan,
                               monotone_constraints='()', n_estimators=250,
                               n_jobs=8, num_parallel_tree=1, predictor='auto',
                               random_state=1, reg_alpha=0, reg_lambda=1,
                               scale_pos_weight=10, subsample=0.8,
                               tree_method='exact', validate_parameters=1,
                               verbosity=None))])
```

Important features of the final model



As we see on the left, most important feature of my model is V36.
V14, V18, V26, V1 are next important variables respectively.

Recommendations

- Since, It is given that the cost of repairing a generator is much less than the cost of replacing it, and the cost of inspection is less than the cost of repair we have focused on Recall scores.
- We used 4 different models to compare their scores. Xgboost is giving the highest score. Even Though it was overfitting on the undersampling tune data, it is fixed with the pipeline.
- V36 is affecting my model more than anything. We should pay attention to this variable to reduce cost for the company.
- V14, V18, V26, V1 are the following important variables for my model. We should focus on them next to reduce cost.
- Majority of the data has no failure, fixing the minor problems will increase the effectiveness of the model.