How fast can a Mercedes Benz car be tested and released?—A Kaggle Competition

I signed up for Kaggle over 10 months ago and I kept myself updated with the different competitions every now and then. I would snoop in and see the data, have a look at the evaluation metrics and also go through some discussions and kernels. But I would never actually DO anything. This competition, however was my first on Kaggle. I teamed up with another friend of mine, Vinayaka Raj. Most of the insights into data cleaning, feature engineering and model tuning were his idea. In this post I would have shared the approach we used.

The main objective of the competition was to decide how fast a car can pass the process of screening so as to reduce the time spent on the test bench and reduce pollution as a result. Since a Mercedes Benz car comes with a wide variety of options, testing each of them to see whether they work as per standards is of paramount importance. Hence, with more number of features being added into a car year after year, the time taken to test one such car increases. As a result, Daimler provided a data set of many such cars having numerous 'anonymous' features to the Kaggle community to come up with the best algorithm that would help them out in this matter.

Evaluation Criteria

First, let us see how submissions were evaluated in this competition. The metric used for evaluation was **Coefficient of Determination** (also called R squared). It is the square of the correlation between the predicted and the actual scores (range 0–1). In simple terms, this metric determines how well or how close the model predicts a real value data after being trained with sample data.

Data Exploration

Before doing so, I want to let you know that we used python and its associated packages (matplotlib, pandas, numpy, sklearn and so on) for this competition.

Now we dive into the data! Let's have a look at it first!

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
1	ID	y	X0	X1	X2	X3	X4	X5	X6	X8	X10	X11	X12	X13	X14
2	0	130.81	k	v	at	a	d	u	j	o	0	0	0	1	0
3	6	88.53	k	t	av	e	d	y	l	o	0	0	0	0	0
4	7	76.26	az	w	n	c	d	x	j	x	0	0	0	0	0
5	9	80.62	az	t	n	f	d	x	l	e	0	0	0	0	0
6	13	78.02	az	v	n	f	d	h	d	n	0	0	0	0	0
7	18	92.93	t	b	e	c	d	g	h	s	0	0	0	0	1
8	24	128.76	al	r	e	f	d	f	h	s	0	0	0	0	1
9	25	91.91	o	l	as	f	d	f	j	a	0	0	0	0	1
10	27	108.67	w	s	as	e	d	f	i	h	0	0	0	0	1
11	30	126.99	j	b	aq	c	d	f	a	e	0	0	0	0	1
12	31	102.09	h	r	r	f	d	f	h	p	0	0	1	0	0
13	32	98.12	al	r	e	f	d	f	h	o	0	0	0	0	1
14	34	82.62	s	b	ai	c	d	f	g	m	0	0	0	0	0

Snapshot of the csv file provided by Kaggle

Each row has a unique ID, a collection of categorical variables(X0—X8) and binary/identity variables(X10—X378) as well as the output 'y'.

Upon analysis, we came across some columns:

1. that showed **non-variance** (same values throughout the column)
2. that were highly **skewed** (few values occurring most of the time (> 95%))
3. that were exact **duplicates**
4. that were **linearly related** to certain other columns (positively or negatively)

We did not come across any columns with missing values. Hence there was no need for imputation.

Data Preprocessing

The main idea behind preprocessing is to counter the aspects we came across in the data exploration phase. So what did we do? Before asking that let us ask '*Why do we do it?*'

1. Reduces dimensionality of the problem in hand.
2. Reduces the possibility of over-fitting (the model tends to learn all available features that are redundant, duplicate or unnecessary).
3. Lesser number of features mean simpler models, lesser memory, reduced training time and faster results.

Now we get back to the question *'What did we do?'*

1. We removed columns showing no variability, meaning columns that contained only one value throughout.
2. We also removed columns whose values occurred in more than 95% in the entire column.
3. We scanned the data set for duplicate occurrences and retained only one column in each case.
4. Columns that showed high correlation were also removed to reduce over-fitting.

Feature Engineering

A feature is a useful piece of information for prediction. Hence, better features lead to better results.

Actually, features (numerical ones) available in the raw dataset can be used for prediction right away. You however, won't achieve the desired level of accuracy needed. What if we could create more features and transform them into something the algorithm can understand? By doing so, we are making things simpler for the model to learn by highlighting the underlying problem.

Impact Encoding

Since the data contained categorical variables, our first job was to transform them into a form understandable by the algorithm. We performed impact encoding, where each categorical variable is replaced by the 'mean' of the occurrence of that variable in that column. Hence a variable with more 'mean' value will have a higher numerical value.

There are many ways of converting categorical variables to numerical ones. This article provides more insights to do so.

Creating new features in categorical variables

We were able to create new features having categorical variables. Here are the first five rows of the train set:

```
In [25]: data.head()
Out[25]:
   ID       y  X0 X1  X2 X3 X4 X5 X6 X8  ...   X375  X376  X377  X378  X379  \
0   0  130.81   k  v  at  a  d  u  j  o  ...      0     0     1     0     0
1   6   88.53   k  t  av  e  d  y  l  o  ...      1     0     0     0     0
2   7   76.26  az  w   n  c  d  x  j  x  ...      0     0     0     0     0
3   9   80.62  az  t   n  f  d  x  l  e  ...      0     0     0     0     0
4  13   78.02  az  v   n  f  d  h  d  n  ...      0     0     0     0     0

   X380  X382  X383  X384  X385
0     0     0     0     0     0
1     0     0     0     0     0
2     0     1     0     0     0
3     0     0     0     0     0
4     0     0     0     0     0

[5 rows x 378 columns]
```

First five rows of the dataframe

Columns X0, X1 and X2 had something in common. They had some variables having two letters (ex: az, ab, bc, etc.) So we thought of creating a new column called X0_club, X1_club and X2_club. Each of these columns would have variables after clubbing those staring with the same letter. For instance: [aa, ab, ac, ……, az] would be clubbed as [a] and [ba, bb, bc, ……, bz] would be clubbed as [b] and so on. The following image shows the dataframe containing the newly created columns:

```
In [27]: data.head()
Out[27]:
   ID       y  X0 X1  X2 X3 X4 X5 X6 X8   ...     X378  X379  X380  X382  \
0   0  130.81   k  v  at  a  d  u  j  o   ...        0     0     0     0
1   6   88.53   k  t  av  e  d  y  l  o   ...        0     0     0     0
2   7   76.26  az  w   n  c  d  x  j  x   ...        0     0     0     1
3   9   80.62  az  t   n  f  d  x  l  e   ...        0     0     0     0
4  13   78.02  az  v   n  f  d  h  d  n   ...        0     0     0     0

   X383  X384  X385  X0_club  X1_club  X2_club
0     0     0     0        k        v        a
1     0     0     0        k        t        a
2     0     0     0        a        w        n
3     0     0     0        a        t        n
4     0     0     0        a        v        n

[5 rows x 381 columns]
```

Notice the three additional columns in the end

New features can be created for numerical variables as well. But since there were plenty of columns already we didn't create more (big mistake).

Note: A keypoint to remember is that the steps performed on the training dataset **MUST** be performed on the test dataset as well. So while doing data preprocessing and feature engineering make sure these steps are done for the test data simultaneously.

Building and training the model

Since our problem is one involving regression, we decided to use Random Forest , which is one among the many ensemble methods available. There are many other algorithms to choose from as well.

We used the RandomForestRegressor function available in the sklearn library to perform regression. The different parameters of this function (number of trees, depth of trees, minimum sample to split and number of features) were decided by running a *for* loop and was validated using cross validation.

Another approach

On visualizing the output variable against the number of observations, we saw that it was highly skewed having a number of outliers as well. We decided to transform this using Box Cox transformation. It transforms a skewed distribution to a normal distribution. We made use of the corresponding function available in scipy for this purpose. However, training the model on the transformed result did not yield an improvement in the validation accuracy

What we could have done?

Here are some techniques/approaches we could have used but did not:

1. Create new features using those variables that are of high importance from the existing data. The importance of a variable can be found out by observing which variable is most dependent on in determining the outcome. Cross validation can help you out in this case.
2. We thought of performing basic **logical** operations (AND, XOR and OR), among the binary variables in order to increase the number of features. But since we had a system with 8GB RAM and given the large number of binary variables (~300), it hanged several times :(. In case this step works for you, care must be taken to ensure duplicate and correlated columns are removed.
3. Use AdaBoost algorithm. Though it is sometimes sensitive to noise and outliers, in most cases it outperforms the other ensemble methods. Care must be taken though, as there is a high tendency to overfit.
4. We could have also used the method of regularization in cohesion with regression. The Lasso function in the sklearn library can help you with that.

Ending Note

Do you have a better way to solve this problem? Or do you any other comments that you would like to share? Please feel free to comment.