Demonstrating the power of feature engineering – Part II: How I beat XGBoost with Linear Regression!

Yes, you read correctly!

In this article, I will show you how to beat the notorious XGBoost with a simple Linear Regression model and the help of a creative feature engineering strategy!

First, if you are familiar with jupyter notebooks and you prefer to see the code used at each step, do not lose your time and go straight to this link: notebook. (**WARNING:** Yes, I know, there are some minor changes between this post and the notebook version (Updated), but the whole strategy and concepts are the same and the results are very very close).

For this experimentation, I will use the popular Kaggle open source dataset "Medical Cost Personal Insurance Forecast". This is a dataset with 7 features and around 1338 rows in it. It is a multivariate regression problem that seems "non linear" at first.

If you have read the first part: Demonstrating the power of feature engineering – Part I, I explained that Feature engineering is often under-estimated in a machine-learning project. Many Data Science practitioners will go through this step too rapidly claiming they "know" their data or that the dataset is "already optimal" with all the relevant business features. Sometimes, going forward with a more complex model or skipping the feature engineering step to hyper-parameters tuning is not always the best move to make.

I will demonstrate this thinking again, but this time, using a more complex and multivariate dataset and at the end, getting a better performance with a linear model (and some ***CREATIVITY***) than with an XGBoost model!

All the python code to my complete solution is available to public.

Table of Contents

Understanding the problem statement and dataset

First of all, let us have a look at the original dataset:

```
dataset = pd.read_csv('insurance_kaggle.csv', sep = ',')
dataset.head(5)
```

	age	sex	bmi	children	smoker	region	charges
0	19	female	27.900	0	yes	southwest	16884.92400
1	18	male	33.770	1	no	southeast	1725.55230
2	28	male	33.000	3	no	southeast	4449.46200
3	33	male	22.705	0	no	northwest	21984.47061
4	32	male	28.880	0	no	northwest	3866.85520

```
dataset.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1338 entries, 0 to 1337
Data columns (total 7 columns):
 #   Column    Non-Null Count  Dtype
---  ------    --------------  -----
 0   age       1338 non-null   int64
 1   sex       1338 non-null   object
 2   bmi       1338 non-null   float64
 3   children  1338 non-null   int64
 4   smoker    1338 non-null   object
 5   region    1338 non-null   object
 6   charges   1338 non-null   float64
dtypes: float64(2), int64(2), object(3)
memory usage: 73.3+ KB
```

```
dataset.shape
```

```
(1338, 7)
```

- **age**: age of primary beneficiary
- **sex**: insurance contractor gender, female, male
- **bmi**: Body mass index, providing an understanding of body, weights that are relatively high or low relative to height, objective index of body weight (kg / m ^ 2) using the ratio of height to weight, ideally 18.5 to 24.9

- **children**: Number of children covered by health insurance / Number of dependents
- **smoker**: Smoking
- **region**: the beneficiary's residential area in the US, northeast, southeast, southwest, northwest.
- **charges**: Individual medical costs billed by health insurance. This is the target that we want to forecast

I took this data dictionary from https://www.kaggle.com/mirichoi0218/insurance. If you go to this link, you can also have an interactive visualization of the dataset.

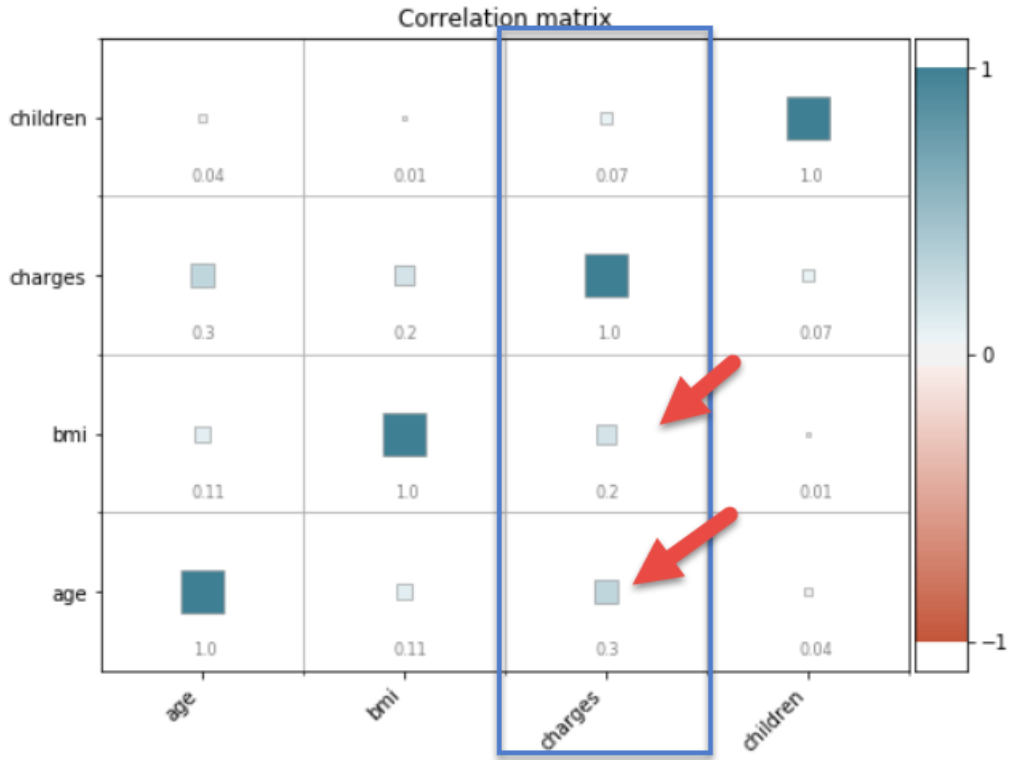
If you want to do some expriments, the dataset is also available here: https://github.com/stedy/Machine-Learning-with-R-datasets

The problem statement is: Can you accurately predict insurance costs (charges)?

Quick exploratory data analysis (EDA)

We can do some Exploratory Data Analysis (EDA) to understand the relationships in the data.
The goal of this article is not to do EDA soo I will go short with the explanations:

Correlation analysis:

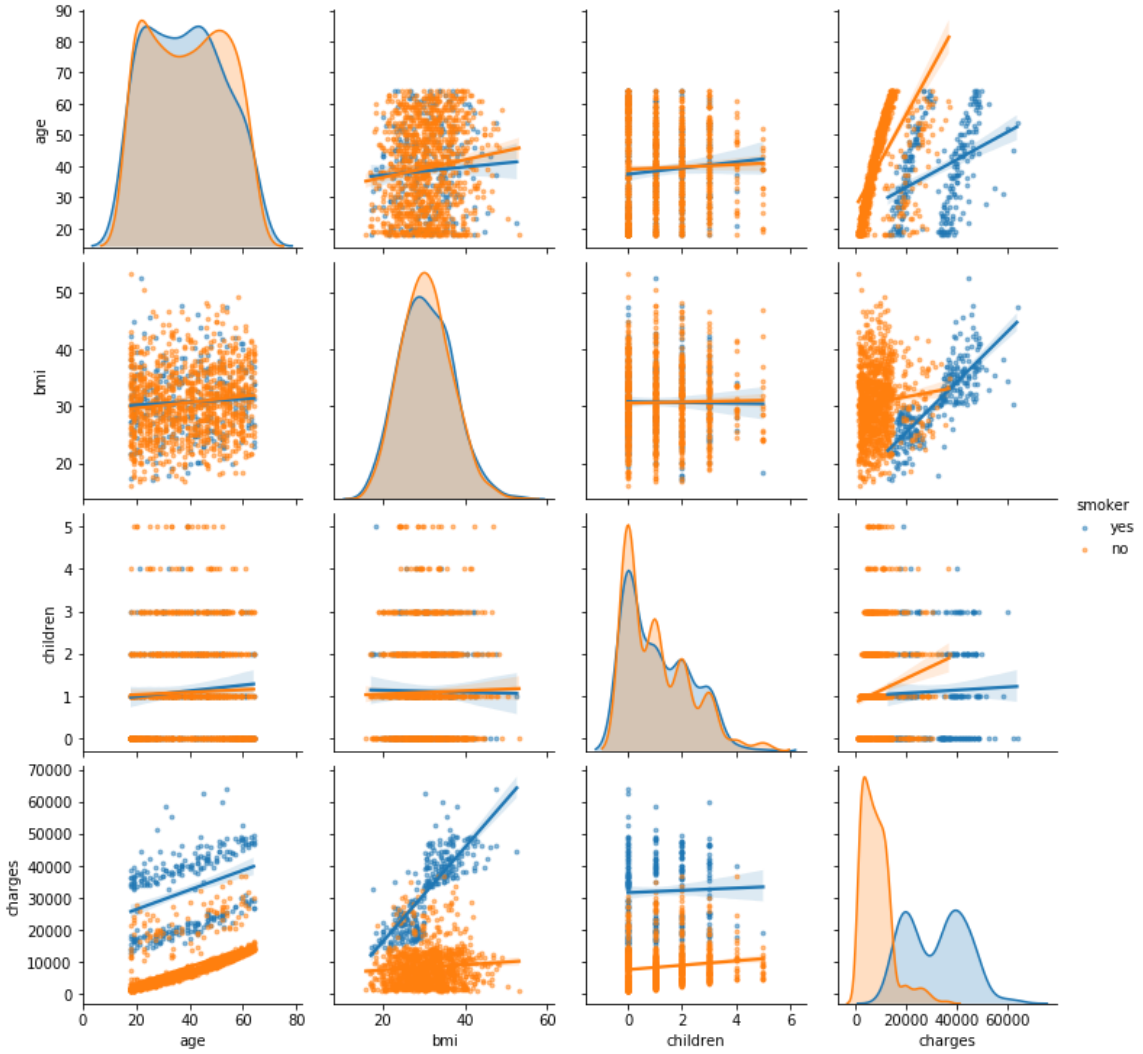


In this correlation matrix (numerical features only), if you look at the predicator column (charges), we already see that features "age" and "bmi" have the biggest positive correlations with charges.
If you are interested in this heatmap vizualisation function, I took the original code from here: github_original_heatmap_function

from there, I added some custom codes of mine to:

- Show the correlation label
- Draw an edgeline around the squares
- Choose the proper figsize
- Choose the correlation label size

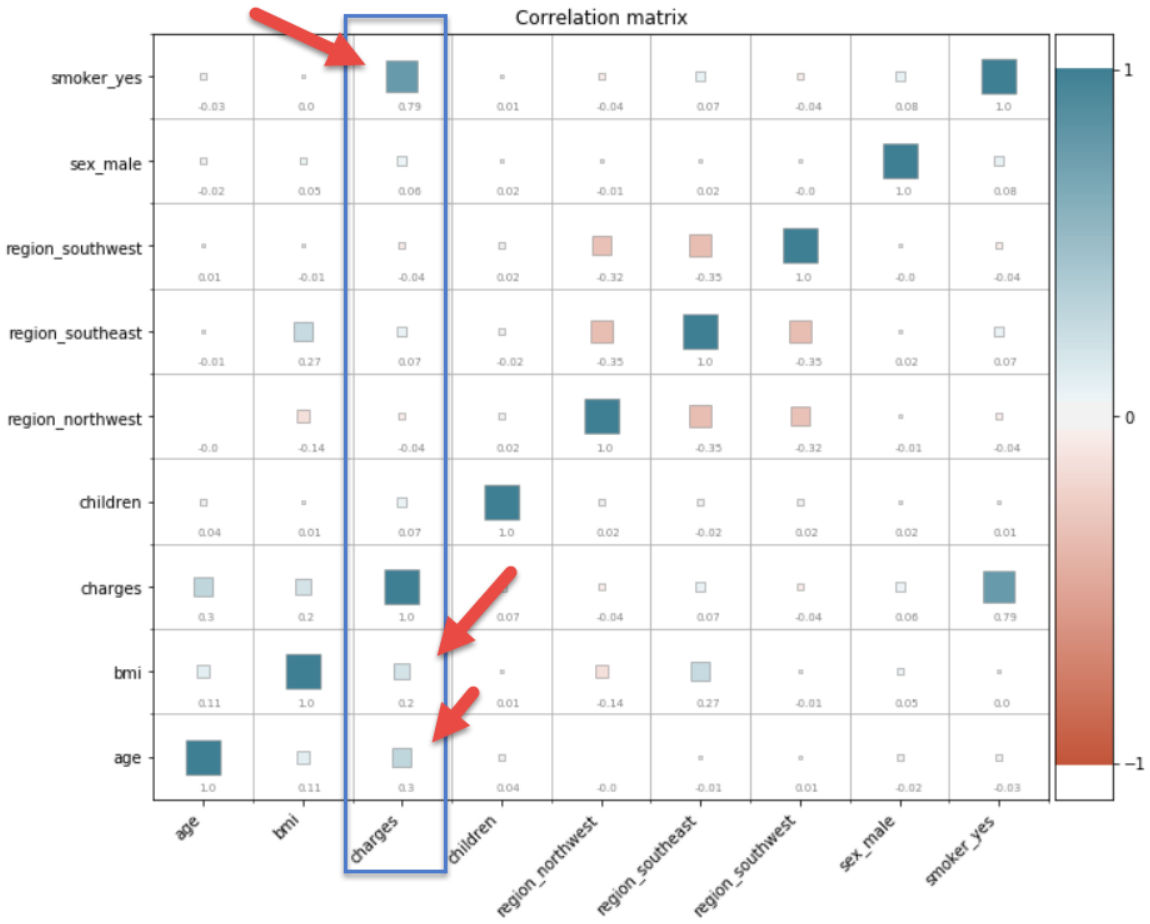
Pairplot with numerical features (Before the dummy encoding):



There is many things happening in this pairplot. *the blue = smoker and orange = non-smoker.

- **Smoker**: We can already see that the smoker categorical feature have a big impact on charges; it makes clear clusters where smoker have a higher charges than non-smoker.
- **Age**: The more aged, the more charges
- **Bmi and smoker**: the more bmi for smokers, the more charges

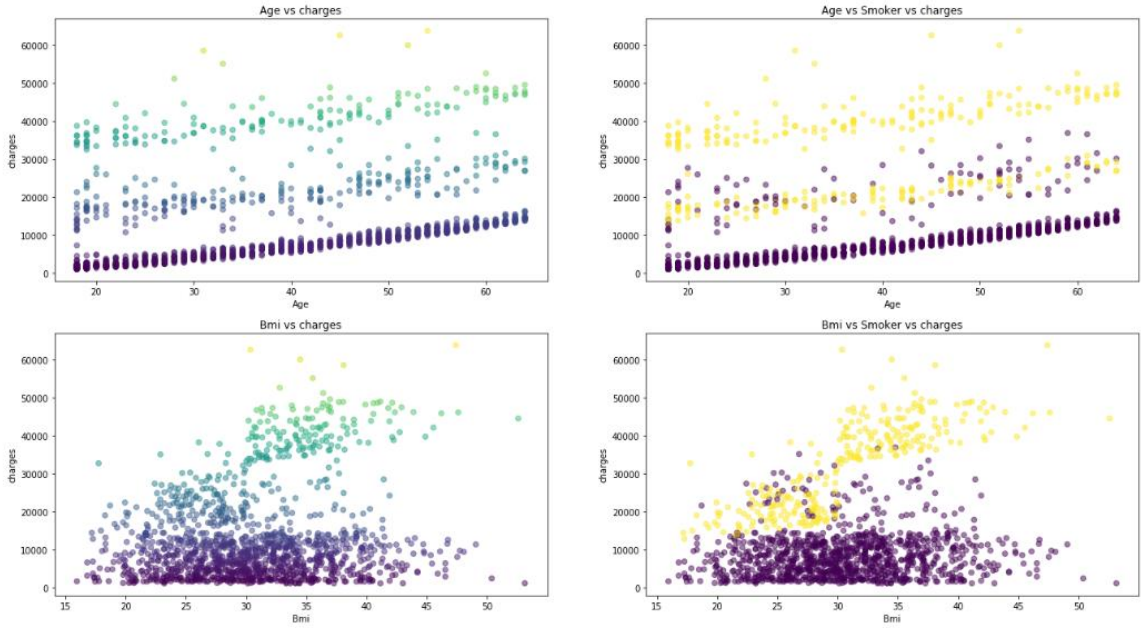
Let us dummy encode the categorical features and plot the correlation matrix again and see if smoker have a big impact



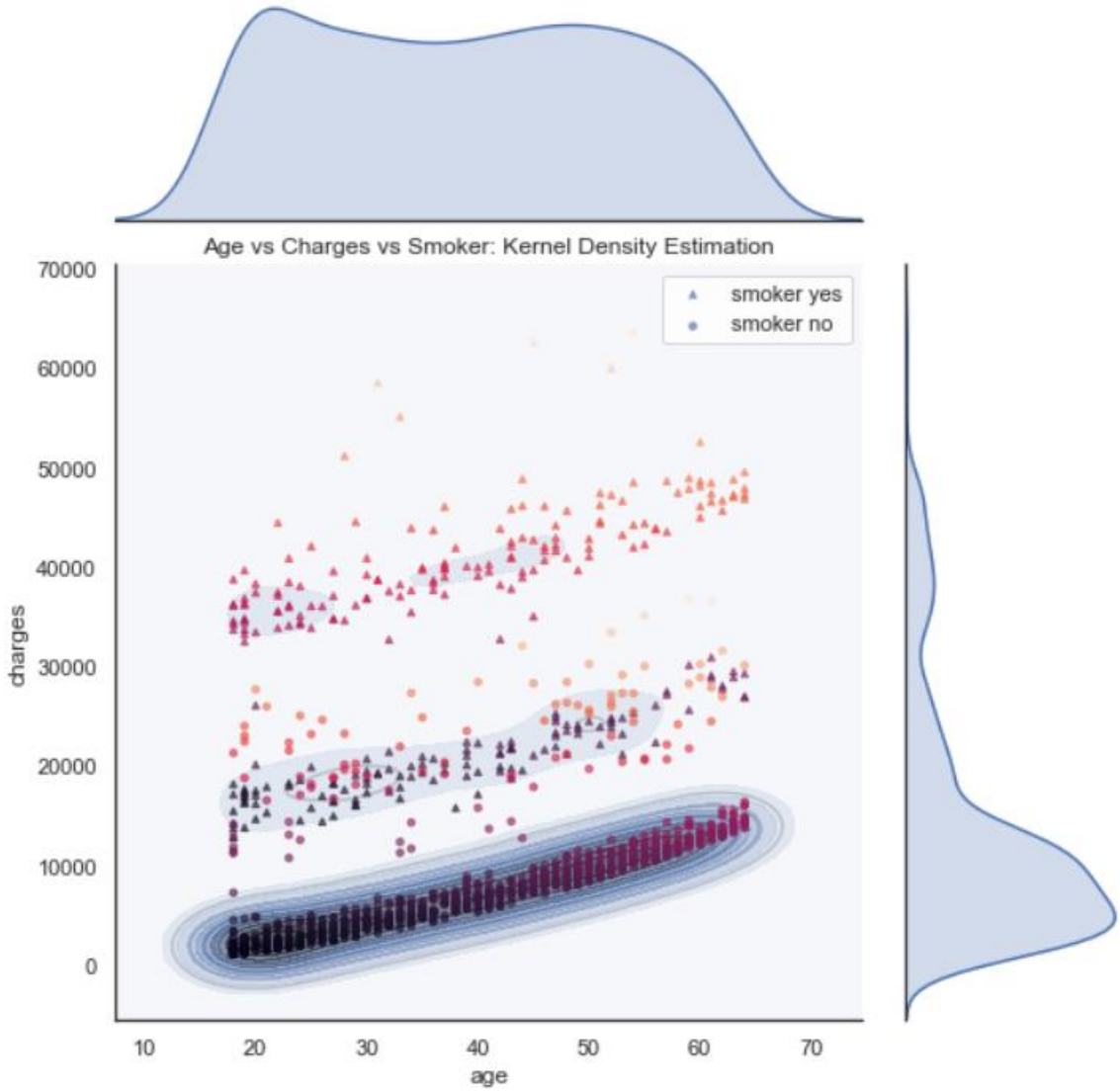
In this new correlation matrix (with the converted categorical features to numerical), we can see that "smoker_yes" have a HUGE positive correlation on charges.

We can also see that there is some multicollinearity between the regions, and bmi vs regions.

Bivariate and Trivariate Analysis:

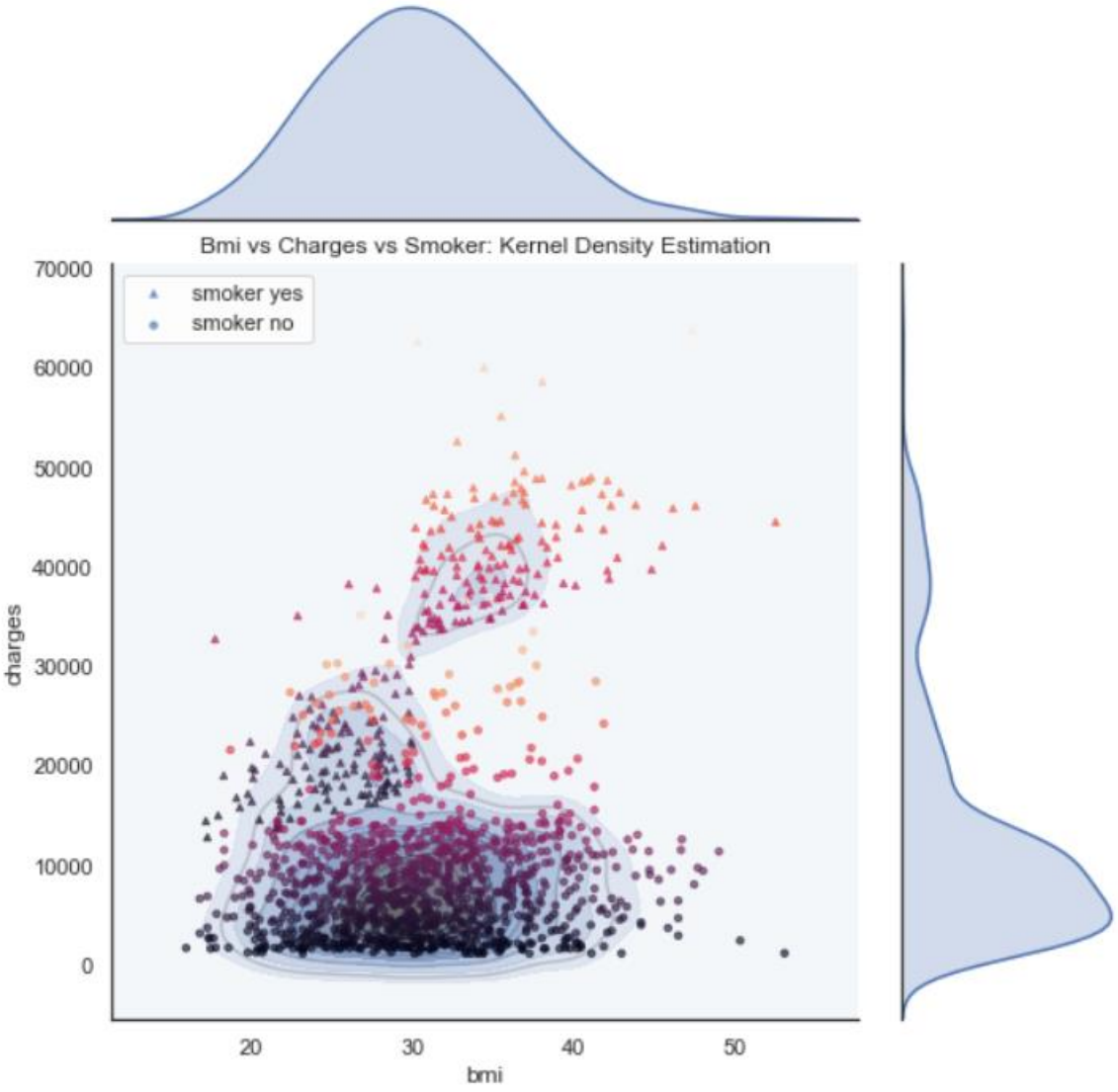


Those graphs help us to see the relations between the 3 more important features in the dataset: Age, Bmi and smoker



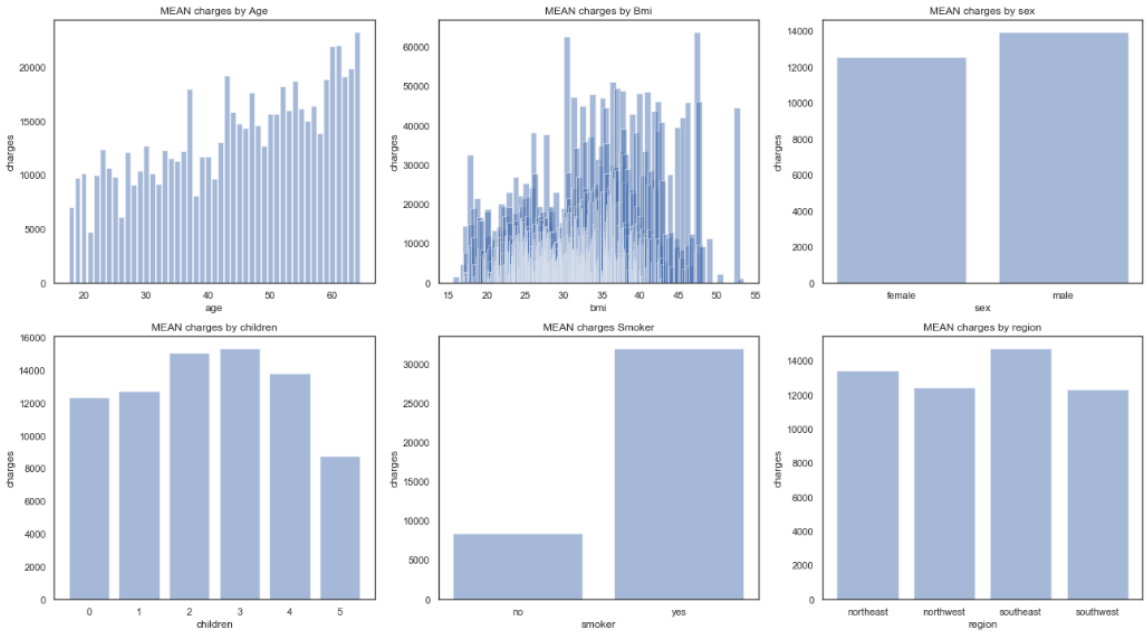
This graph give us a quick information about the data density between Age, Charges and Smoker (triangles = smoker_yes).

- We can see that the majority of the data is in the bottom cluster, low charges and non-smoker.
- The top cluster is only smoker with high charges,
- The middle cluster is a mix of smoker and non-smokers with moderate charges. We can already say that the model will have some difficulties with the middle cluster as it is mixed up!



If we do the same graph with bmi,

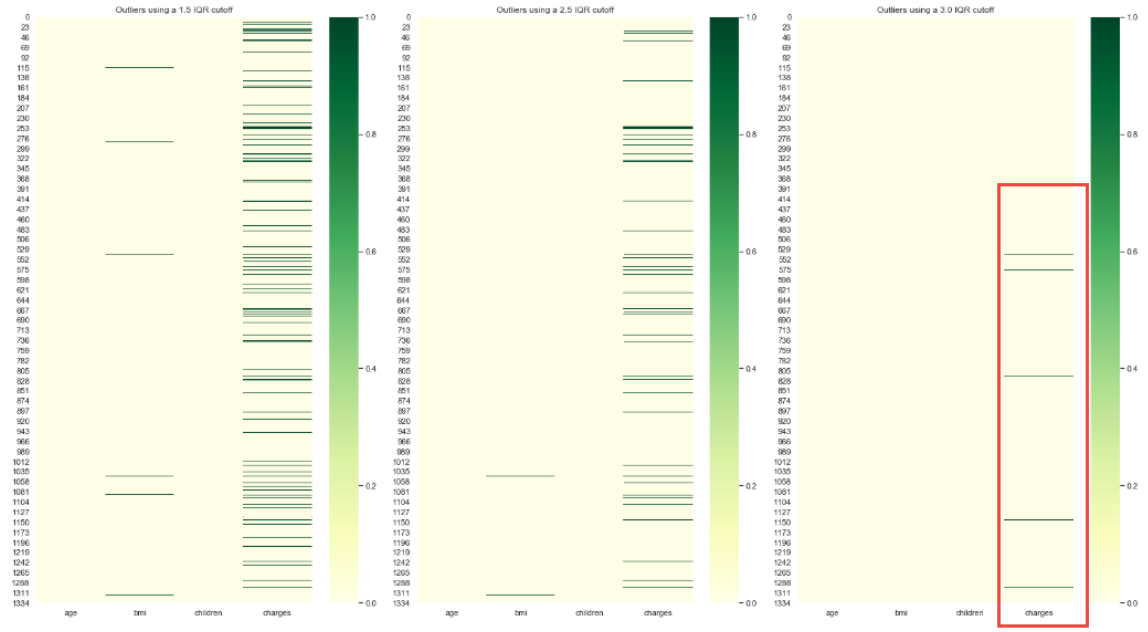
- The top cluster have only smokers, with high charges and moderate-high bmi
- the middle cluster is again, a mixed-up of smokers and non-smokers with moderate charges and low-moderate bmi
- The bottom cluster is only non-smokers, with all kind of bmi



This graph give us more information about the mean charges:

- Mean charges goes up with ages
- Mean charges goes up than goes down with bmi
- Mean charges is a little bit higher for male
- Mean charges is a little bit higher when 2 and 3 childrens
- Mean charges is a lot higher when smoker = Yes
- Mean charges is a little bit higher for the southeast region

Outlier Analysis:

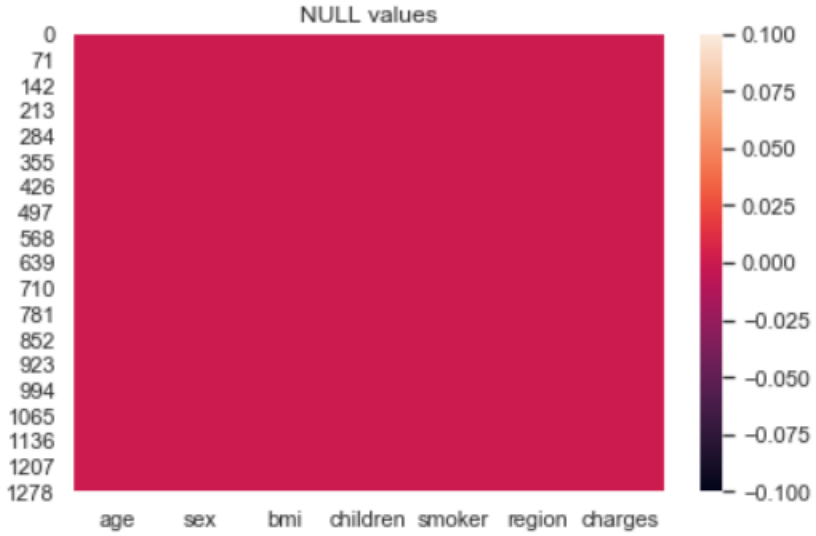


This graph shows us the outliers, by numerical features. We can see that with the IQR method, there are few rows that are more than 3 IQR that we will clean later in the dataset.

NULL analysis:

```
#NULL ANALYSIS
print(dataset.isnull().sum())
sns.heatmap(dataset.isnull(), cbar=True)
plt.title('NULL values')
plt.tight_layout()
```

```
age         0
sex         0
bmi         0
children    0
smoker      0
region      0
charges     0
dtype: int64
```



Finally, the NULL analysis: Lucky us! No NULL values to clean!

To conclude this quick EDA step;

- smoker_yes seems to be the most correlated features with charges. When smoker_yes = 1, charges seems to be a lot higher. This is a discrimant feature!
- The age and bmi follow smoker_yes as the most correlated features with charges. When the age is higher, the charges seems to be higher. There are also some evident data clusters when we look at charges by age and by bmi.
- The male sex seems to have slightly higher charges than female.
- The southeast region seems to have slightly higher charges than other regions.
- The rest of features seems to have a minor effect on the charges.

- They seems to have some multicolinearity between the regions and the bmi features.

Data preparation

We have some basic cleaning to do before going anywhere with it.

For that, I will use a custom tool package that I have built (The code is available in my github).

The DataPreparator class from fboost.py will automaticly do these few steps:

- replace_null (numerical = Mean, categorical = Mode)
- drop_duplicate_columns
- drop_duplicate_rows (without the predicator)
- drop_constant_columns
- drop_outliers (we can change the strategy but here, we will use an IQR >= 3)
- categorical_encoding (we can change the strategy but here, we will use basic dummy encoding)

If you want to know more about how it is done, you can have a look at the fboost.py, in the DataPreparator class code section.

IMPORTANT NOTE: This fboost.py package is a quick implementation done for this demonstration; it is not bulletproof and may contains many bugs! Be aware if you use it for other purpose than this experimentation!

The last step we have to do is to encode our categorical features to numerical. We could have use a more sophisticated technic like the CatBoost or the Target Encoder, but we will stay simple for now, with a basic dummy encoding technic. If you want to know more about Categorical encoding technics and tools for python, go to this link: http://contrib.scikit-learn.org/categorical-encoding/targetencoder.html

```
X_train.head(5)
```

	age	bmi	children	region_northwest	region_southeast	region_southwest	sex_male	smoker_yes
954	34	27.835	1	1	0	0	1	1
951	51	42.900	2	0	1	0	1	1
335	64	34.500	0	0	0	1	1	0
957	24	26.790	1	1	0	0	1	0
454	32	46.530	2	0	1	0	1	0

As we can see, the categorical features are now numerical with binary values.

If you are new to this and want to know what is dummy encoding or why we discarded one dummy variable for each categorical variables, go to this link (Quick explanation): dummy_encoding_link

We have now 9 features, all numerical and ready:

```
 #   Column            Non-Null Count  Dtype
---  ------            --------------  -----
 0   age               1332 non-null   int64
 1   bmi               1332 non-null   float64
 2   children          1332 non-null   int64
 3   charges           1332 non-null   float64
 4   sex_male          1332 non-null   uint8
 5   smoker_yes        1332 non-null   uint8
 6   region_northwest  1332 non-null   uint8
 7   region_southeast  1332 non-null   uint8
 8   region_southwest  1332 non-null   uint8
```

Linear Regression vs XGBoost BEFORE feature engineering:

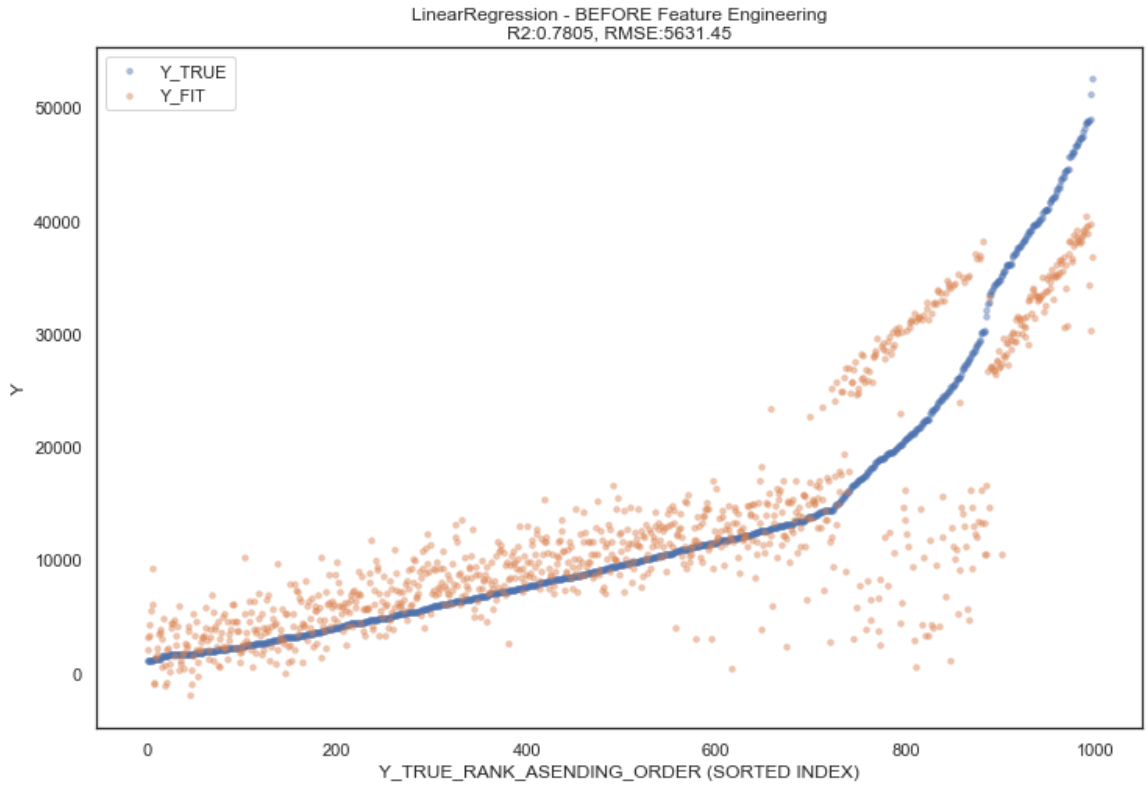
Linear Regression Model:

```
"""
###############################################
# 4-LINEAR REGRESSION MODEL WITHFEATURE ENGINEERING #
###############################################
"""

model = LinearRegression()
model.fit(X_train_scaled, y_train)
r2_linreg_before = model.score(X_test_scaled, y_test)
rmse_linreg_before = np.sqrt(mean_squared_error(y_test, model.predict(X_test_scaled)))
#print('R2:' + str(round(r2_linreg_before, 4)) + ', RMSE:' + str(round(rmse_linreg_before, 2)))
```

```
print('LinearRegression R2 (Before feature engineering): ' + str(round(r2_linreg_before, 4)))
print('LinearRegression RMSE (Before feature engineering): ' + str(round(rmse_linreg_before, 4)))
```

```
LinearRegression R2 (Before feature engineering): 0.7805
LinearRegression RMSE (Before feature engineering): 5631.4485
```



This multivariate regression visualization trick is explained in detail with python codes in one of my previous article: [Visualization trick for multivariate regression problems]

Just by looking at the graph, we can spot a couple of problems (underfitting) in this Multivariate Linear Regression model.

An instinctive basic solution to this problem would be: Let's use a non-linear model instead! Let's try with the notorious XGBoost and see how it goes. We can cheat and use GridSearch to find an optimal hyper-parameters setting for the model:

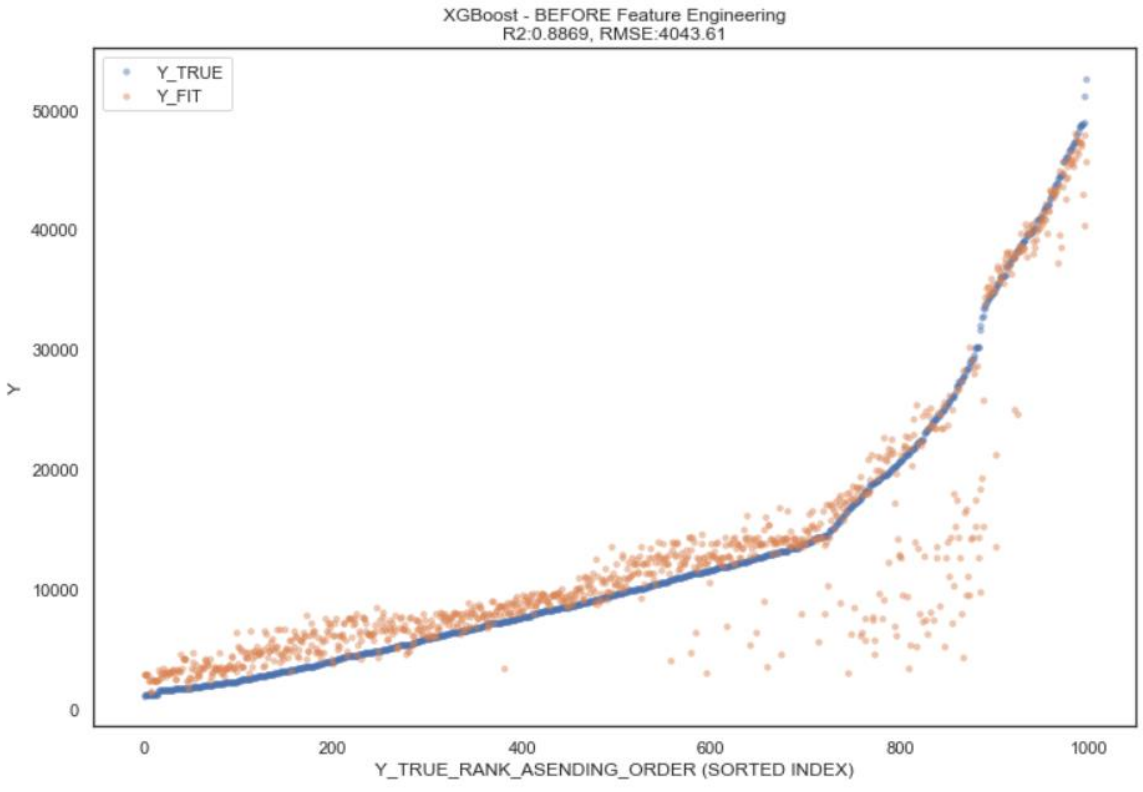
```
"""
#############################################
# 4-XGBOOST MODEL WITHOUT FEATURE ENGINEERING #
#############################################
"""

#### GRID SEARCH
# A parameter grid for XGBoost
params = {
        'min_child_weight': [1, 5, 10],
        'gamma': [0.5, 1, 1.5, 2, 5],
        'subsample': [0.6, 0.8, 1.0],
        'colsample_bytree': [0.6, 0.8, 1.0],
        'max_depth': [3, 4, 5],
        'n_estimators': [40,50,80,100],
        'learning_rate': [0.1, 0.01],
        'random_state': [0]
        }
model = XGBRegressor()

grid = GridSearchCV(estimator=model,
                    param_grid=params,
                    scoring='neg_root_mean_squared_error',
                    n_jobs=8,
                    cv=3,
                    verbose=3 )
grid.fit(X_train,y_train)
print(grid.best_estimator_)

#With GridSearch Best Hyper-parameters
model = XGBRegressor(min_child_weight = 10,
                     gamma = 0.5,
                     subsample = 1.0,
                     learning_rate = 0.1,
                     colsample_bytree = 1.0,
                     max_depth = 3,
                     n_estimators = 50,
                     random_state = 0)
model.fit(X_train, y_train)
r2_xgb_before = model.score(X_test, y_test)
rmse_xgb_before = np.sqrt(mean_squared_error(y_test, model.predict(X_test)))
print('R2:' + str(round(r2_xgb_before, 4)) + ', RMSE:' + str(round(rmse_xgb_before, 2)))
```

```
Fitting 3 folds for each of 3240 candidates, totalling 9720 fits
[Parallel(n_jobs=8)]: Using backend LokyBackend with 8 concurrent workers.
[Parallel(n_jobs=8)]: Done  16 tasks      | elapsed:    1.9s
[Parallel(n_jobs=8)]: Done 272 tasks      | elapsed:    4.0s
[Parallel(n_jobs=8)]: Done 912 tasks      | elapsed:    9.0s
[Parallel(n_jobs=8)]: Done 1808 tasks      | elapsed:   16.1s
[Parallel(n_jobs=8)]: Done 2960 tasks      | elapsed:   25.2s
[Parallel(n_jobs=8)]: Done 4368 tasks      | elapsed:   37.6s
[Parallel(n_jobs=8)]: Done 6032 tasks      | elapsed:   53.7s
[Parallel(n_jobs=8)]: Done 7952 tasks      | elapsed:  1.2min
[Parallel(n_jobs=8)]: Done 9705 out of 9720 | elapsed:  1.5min remaining:    0.0s
[Parallel(n_jobs=8)]: Done 9720 out of 9720 | elapsed:  1.5min finished
XGBRegressor(base_score=0.5, booster='gbtree', colsample_bylevel=1,
             colsample_bynode=1, colsample_bytree=1.0, gamma=0.5,
             importance_type='gain', learning_rate=0.1, max_delta_step=0,
             max_depth=3, min_child_weight=10, missing=None, n_estimators=50,
             n_jobs=1, nthread=None, objective='reg:linear', random_state=0,
             reg_alpha=0, reg_lambda=1, scale_pos_weight=1, seed=None,
             silent=None, subsample=1.0, verbosity=1)
R2:0.8869, RMSE:4043.61
```



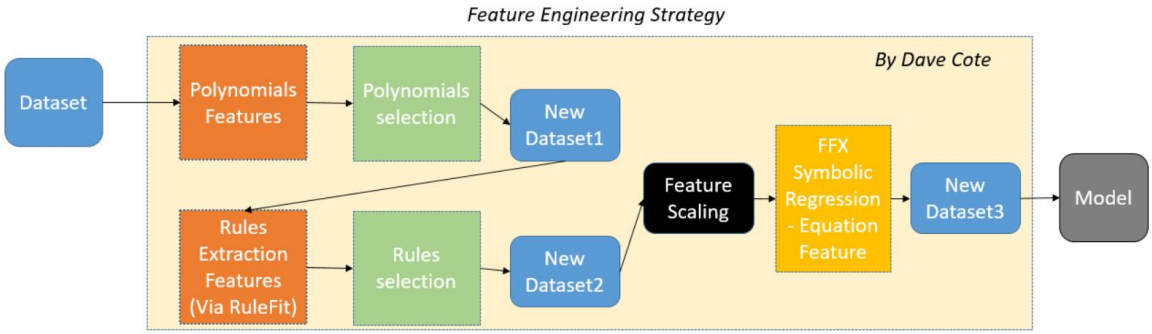
Wow! That is going to be hard to beat! XGBoost performed a lot better than the Linear Regression model on this round! The scores are:

```
XGBoost R2 (Before feature engineering): 0.8869
XGBoost RMSE (Before feature engineering): 4043.6107
```

This sound like Mission Impossible! how the hell are we going to engineer good features to beat those results with a simple Linear Regression model ???

We need at least 28.1959% of RMSE improvement… that seems HUGE!!!

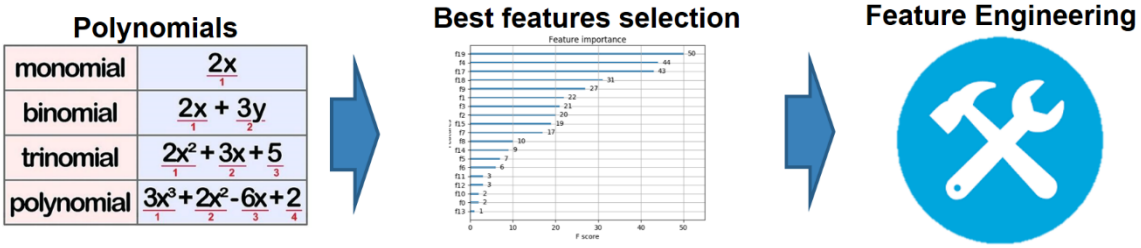
Implementing the feature engineering strategy



This diagram is very important. This is the whole Feature Engineering strategy that we will use to beat the XGBoost regression model.

Process description:
1 - Polynomials features are created. Second-degree polynomials is enough. Then, we do a feature selection step to keep only the better ones.

To be clear, we pass **ONLY** the polynomials features (Not the original ones) in the feature selection automated process.



2 - Rules features are created using a RuleFit Ensembe model. Then, we do a feature selection step to keep only the better rules.

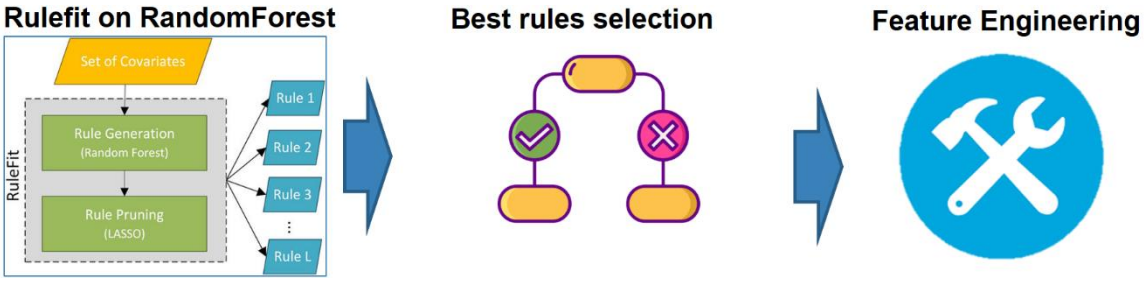
To be clear, we pass **ONLY** the rules features (Not the original ones) in the feature selection automated process.

“RuleFit is not new. It seems to have first appeared in a paper by Jerome Friedman and Bogdan Popescu of Stanford in October 2005 titled “Predictive Learning Via Rule Ensembles””.

RuleFit is a very interesting underrated model and you can read more on it here: RuleFit_important
The original paper: RuleFit_scientific_paper

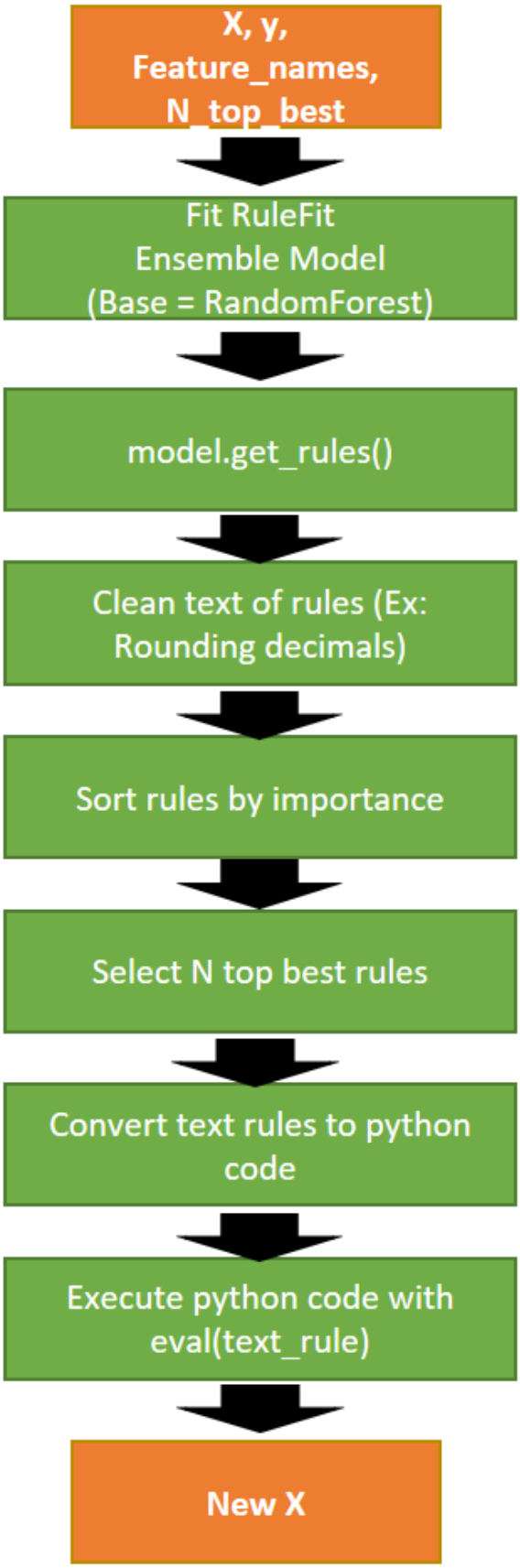
To be short, RuleFit is a Lasso linear model applied on extracted rules from a tree model or an ensemble tree model. We will use this model to extract the best rules of a Random Forest model and then, add those rules features to our dataset.

The python package I used for the RuleFit model is available on this github: RuleFit_github



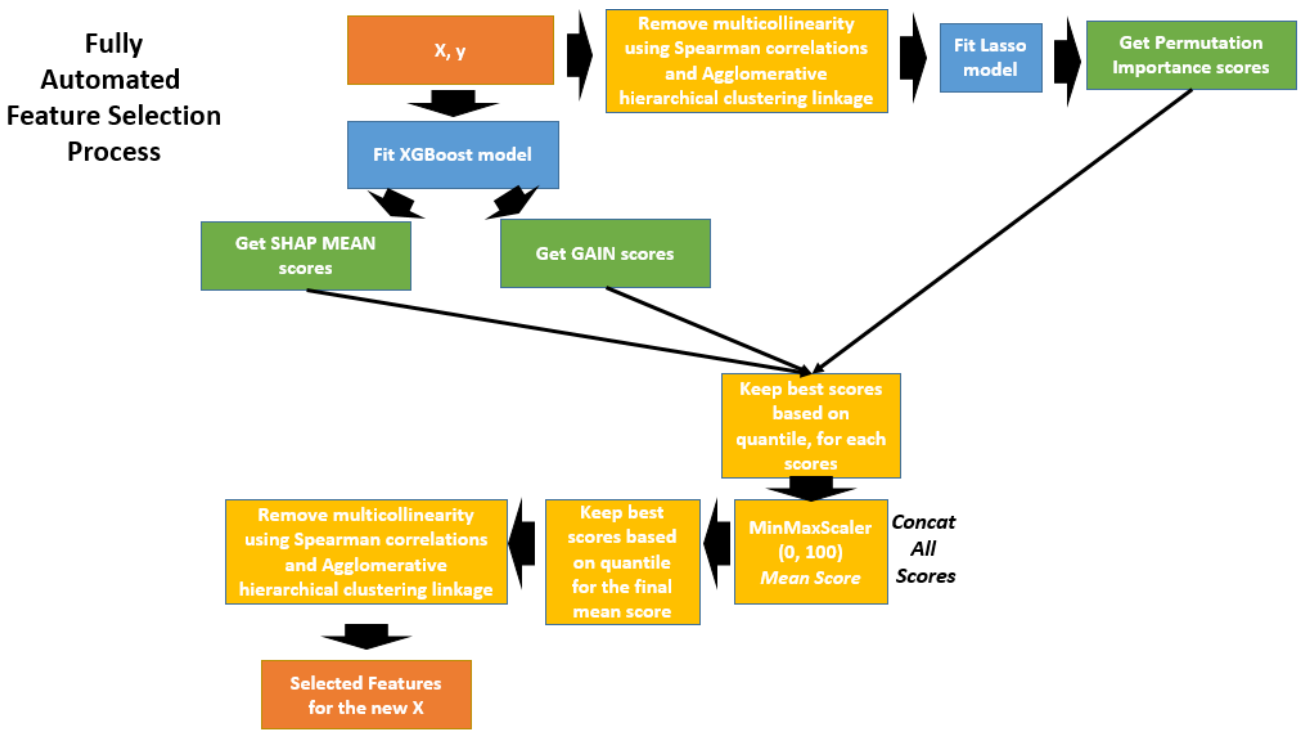
The process to extract rules from RuleFit model and then convert them to python code to produce the new features is:

Fully Automated Rules Extraction Process



Each rules has been converted to python code to create features that will be binary indicators (1 if the rule is applied on the row). I will show an example later.

The feature selection technique used is custom. Here is the feature selection process implemented in fboost.py:



There is a lot of thing happening in this automated feature selection process. If you are a new to any of these concepts, I invite you to read this:

- Shapley values: shap_kdnuggets, shap
- Multicollinearity: multicollinearity
- Permutation Scores: Permutation_Scores
- Feature Importance Gain scores: Feature_Importance_Tree
- MinMax Scaling: MinMaxScaler
- Agglomerative Hierarchical Clustering: permutation_multicollinear, agglomerative_HC

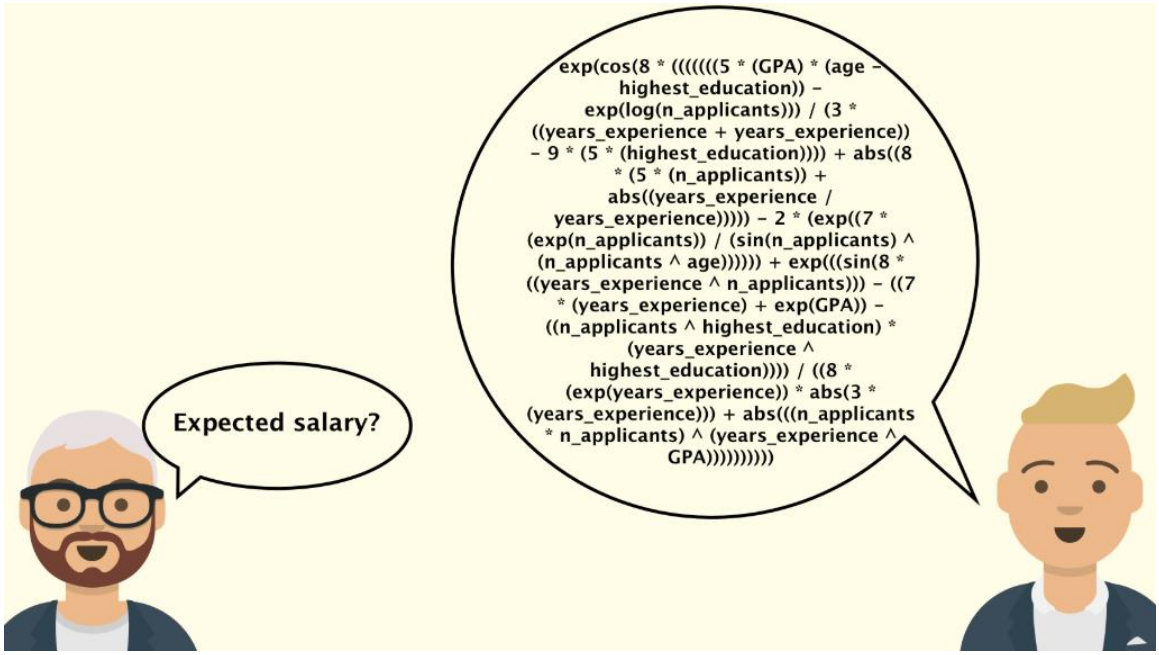
3 – FFX Symbolic Regression model. The last Feature Engineering step is the FFX Symbolic Regression.

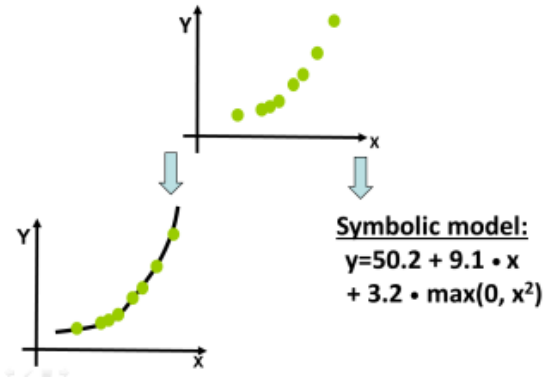
From wikipedia: Symbolic_Regressoin

"Symbolic regression is a type of regression analysis that searches the space of mathematical expressions to find the model that best fits a given dataset, both in terms of accuracy and simplicity.

No particular model is provided as a starting point to the algorithm. Instead, initial expressions are formed by randomly combining mathematical building blocks such as mathematical operators, analytic functions, constants, and state variables."
A great explanation is available here: Symbolic_Regressoin2

"To be short, Symbolic Regression is a type of regression analysis that searches the space of mathematical expressions to find the model that best fits a given dataset"





As explained here: FFX

FFX stands for Fast Function Extraction.

FFX is a technique for symbolic regression, to induce whitebox models given X/y training data.

It does Fast Function Extraction. It is:

- Fast - runtime 5-60 seconds, depending on problem size (1GHz cpu)

- Scalable - 1000 input variables, no problem!
- Deterministic - no need to "hope and pray".

If you ignore the whitebox-model aspect, FFX can be viewed as a regression tool. It's been used this way for thousands of industrial problems with 100K+ input variables. It can also be used as a classifier (FFXC), by wrapping the output with a logistic map. This has also been used successfully on thousands of industrial problems.

The python package I use for the Feature Enginnering process is: FFX_github

We will use this technic to extract the optimal function that explained charges from the feature in our dataset. Then, with the function given by the FFX model, we will create a last feature that is the result of applying the function on any new incoming data.

```
###############WITH FEATURE ENGINEERING

# Reload the dataset
dataset = pd.read_csv('insurance_kaggle.csv', sep = ',')
y_colname = 'charges'

# SPLIT DATA
X = dataset[dataset.columns.difference([y_colname])]
y = dataset[[y_colname]]
feature_names = list(dataset.columns.difference([y_colname]))

X_train, X_test, y_train, y_test = train_test_split(X, y,
                                                    test_size=0.25,
                                                    stratify = X['smoker'],
                                                    random_state=0)

# DATA PREP
data_prep = DataPreparator(outliers_strategy = 'IQR',
                           outliers_cutoff = 3,
                           encoding_strategy = 'dummy',
                           drop_duplicate_rows = True)
X_train, y_train = data_prep.fit_transform(X_train, y_train)
X_test = data_prep.transform(X_test, y_test)
```

Now, with the use of my custom fboost.py FeatureBoosterRegressor class, let us automate the feature engineering process of polynomials, rules extraction and features selection:

```
# FEATURE ENGINEERING - 1/2 - Polynomials + Rules extraction
fboost = FeatureBoosterRegressor(base_model = RandomForestRegressor(criterion='friedman_mse',
                                                    max_depth=5,
                                                    max_features=None,
                                                    max_leaf_nodes=2,
                                                    min_samples_leaf=1,
                                                    verbose=0,
                                                    n_estimators = 850,
                                                    warm_start=True,
                                                    random_state = 0),
                                 max_rules = 2800,
                                 n_best_rules = 35,
                                 original_features_selection= False,
                                 selection_strategy = 'severe',
                                 quantile_cutoff = 0.83,
                                 alpha = 89,
                                 scaler = 'Standard',
                                 random_state = 0)

# FIT FEATURE ENGINEERING FOR TRAIN DATA
X_train, rules = fboost.fit_transform(X_train, y_train)

# TRANSFORM FEATURE ENGINEERING FOR TEST DATA
X_test = fboost.transform(X_test)
```

Let’s look at the new dataset:

```
#LET HAVE A LOOK AT THE NEW FEATURES WE JUST CREATED
X_train.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 997 entries, 954 to 993
Data columns (total 14 columns):
 #   Column            Non-Null Count  Dtype  
---  ------            --------------  -----  
 0   age               997 non-null    int64  
 1   bmi               997 non-null    float64
 2   children          997 non-null    int64  
 3   region_northwest  997 non-null    int32  
 4   region_southeast  997 non-null    int32  
 5   region_southwest  997 non-null    int32  
 6   sex_male          997 non-null    int32  
 7   smoker_yes        997 non-null    int32  
 8   bmi_smoker_yes    997 non-null    float64
 9   age2              997 non-null    float64
 10  age_children      997 non-null    float64
 11  RULE_EXTRACT_1    997 non-null    float64
 12  RULE_EXTRACT_18   997 non-null    float64
 13  RULE_EXTRACT_8    997 non-null    float64
dtypes: float64(7), int32(5), int64(2)
memory usage: 97.4 KB
```

We can see that we have added 3 Polynomials features:

- bmi_smoker_yes (bmi * smoker_yes)
- age2 (age * age)
- age_children (age * children)

We have also added Rules features:

```
selected_rules = fboost.rules_df_.loc[fboost.rules_df_['SELECTED'] == 1]
for row in range(0, len(selected_rules)):
    print(selected_rules.iloc[row, 0] + ': ' + selected_rules.iloc[row, 1])
```

```
RULE_EXTRACT_1: bmi_smoker_yes > 30.108 & bmi_smoker_yes > 26.05
RULE_EXTRACT_8: bmi_smoker_yes <= 27.721 & bmi_smoker_yes <= 8.591 & age2 <= 2652.5
RULE_EXTRACT_18: bmi_smoker_yes <= 26.05
```

Where did I get the rules ? After the FeatureBoosterRegressor FIT, we can have access to the rules dataframe:

```
fboost.rules_df_
```

	FEATURE_NAME	RULE	PYTHON_CODE	SELECTED
0	RULE_EXTRACT_1	bmi_smoker_yes > 30.108 & bmi_smoker_yes > 26.05	((data['bmi_smoker_yes'] > 30.108) & (data['b...	1
1	RULE_EXTRACT_2	age_children <= 19.5 & bmi_smoker_yes <= 30.10...	((data['age_children'] <= 19.5) & (data['bmi_...	0
2	RULE_EXTRACT_3	bmi_smoker_yes <= 30.108 & bmi_smoker_yes <= 1...	((data['bmi_smoker_yes'] <= 30.108) & (data['...	0
3	RULE_EXTRACT_4	bmi_smoker_yes > 28.305 & age > 27.5 & bmi > 3...	((data['bmi_smoker_yes'] > 28.305) & (data['a...	0
4	RULE_EXTRACT_5	bmi_smoker_yes <= 29.812 & age_children > 53.5...	((data['bmi_smoker_yes'] <= 29.812) & (data['...	0
5	RULE_EXTRACT_6	bmi_smoker_yes <= 28.433 & age <= 41.5	((data['bmi_smoker_yes'] <= 28.433) & (data['...	0
6	RULE_EXTRACT_7	bmi_smoker_yes <= 30.107 & age2 <= 1682.0 & bm...	((data['bmi_smoker_yes'] <= 30.107) & (data['...	0
7	RULE_EXTRACT_8	bmi_smoker_yes <= 27.721 & bmi_smoker_yes <= 8...	((data['bmi_smoker_yes'] <= 27.721) & (data['...	1
8	RULE_EXTRACT_9	age_children > 58.5 & bmi_smoker_yes <=	((data['age_children'] > 58.5) &	0

Now, the last part of feature engineering: FFX Symbolic regression:

```
# FEATURE ENGINEERING - 2/2 - FFX Symbolic Regression, Extraction the function equation and apply it on train and test !
FFX = FFXRegressor()
FFX.fit(X_train_scaled.values, y_train.values.ravel())
print("Score:", FFX.score(X_test_scaled.values, y_test.values.ravel()))
```

```
Score: 0.8759002691138066
```

The score of the equation alone to explain the charges got an R2 > 0.80, wich is not bad at all and better than the original Linear Regression !

If we look at the function created:

```
1.32e4 + 3.62e6*max(0,X10-2.84) * max(0,0.00720-X0) + 2.07e6*max(0,X10-1.49) * max(0,-0.592-X9) - 1.64e6*max(0,X10-1.49) * max(0,-0.427-X0) - 8.00e5*max(0,X10-2.17) * max(0,-0.427-X0) - 2.46e5*max(0,X10-2.17) * max(0,X0-1.31) + 1.86e5*max(0,X10-2.84) * max(0,X0-1.31) + 1.62e5*max(0,X10-1.49) * max(0,0.00720-X0) - 1.62e5*max(0,X2-2.70) * max(0,-0.592-X9) + 1.31e5*max(0,X10-1.49) * max(0,X9-1.63) + 1.30e5*max(0,X10-1.49) * max(0,X0-1.31) - 1.25e5*max(0,X2-2.70) * max(0,X0-0.876) + 1.23e5*max(0,X2-1.58) * max(0,X0-0.876) - 1.02e5*max(0,X10-2.17) * max(0,X0-0.876) - 1.02e5*max(0,X10-1.49) * max(0,-0.148-X9) - 8.43e4*max(0,X10-2.17) * max(0,X9-1.63) + 7.69e4*max(0,X2-1.58) * max(0,-0.592-X9) - 6.52e4*abs(X10) * max(0,X9-1.63) + 5.84e4*max(0,X10-2.84) * max(0,X0-0.876) + 5.61e4*max(0,X2-2.70) * max(0,0.00720-X0) - 5.34e4*max(0,X2-1.58) * max(0,-0.427-X0) - 5.19e4*max(0,X0+0.427) * max(0,0.00720-X0) - 5.16e4*max(0,X8-2.01) * max(0,X0-1.31) + 4.76e4*max(0,X8-2.01) * max(0,X9-1.63) + 4.50e4*max(0,X10-2.84) * max(0,X2-2.70) + 4.48e4*max(0,X0-0.00720) * max(0,X2-2.70) + 4.11e4*abs(X10) * max(0,X0-1.31) - 3.71e4*abs(X10) * max(0,X0-0.876) + 3.57e4*max(0,X8-2.01) + 3.42e4*max(0,X10-1.49) * max(0,X0-0.876) + 3.24e4*max(0
```

,X2-2.70) * max(0,-0.148-X9) - 2.51e4*max(0,X9-1.63) * max(0,X0-1.31
) - 2.34e4*max(0,X9-0.296) * max(0,X2-2.70) - 2.24e4*max(0,X0+0.427)
* max(0,X2-2.70) + 2.23e4*max(0,X9-1.63) * max(0,X0-0.876) + 2.21e4*
abs(X10) * max(0,-0.592-X9) + 2.10e4*max(0,X2-1.58) * max(0,-0.148-X
9) - 1.92e4*max(0,X10-2.84) * X11 + 1.90e4*max(0,X10-3.51) + 1.89e4*
max(0,X0-0.876) - 1.88e4*max(0,X0-0.00720) * max(0,X0-1.31) - 1.88e4
*max(0,X10-2.84) * abs(X9) + 1.86e4*max(0,X0+0.427) * max(0,X8-3.13)
+ 1.77e4*max(0,X10-2.17) * max(0,0.345-X8) - 1.75e4*max(0,X0-0.00720
) * max(0,X8-2.01) - 1.74e4*max(0,X2-1.58) * abs(X9) + 1.73e4*max(0,
X10-2.17) * max(0,X8-2.01) - 1.66e4*abs(X10) * max(0,-0.427-X0) - 1.
60e4*abs(X9) * max(0,X2-2.70) + 1.57e4*max(0,-0.148-X9) * max(0,-0.5
92-X9) - 1.54e4*max(0,X9-0.296) * max(0,X8-2.01) - 1.47e4*max(0,X10-
2.17) * max(0,X2-2.70) - 1.47e4*max(0,X0-0.876) * max(0,X0-1.31) + 1
.46e4*X11 * max(0,X0-1.31) + 1.44e4*X7 * max(0,X9-1.63) - 1.41e4*X11
* max(0,X9-1.63) + 1.39e4*abs(X9) * max(0,-0.592-X9) + 1.31e4*abs(X9
) * max(0,X8-2.01) - 1.31e4*max(0,0.345-X8) * max(0,X9-0.741) - 1.30
e4*max(0,X10-2.84) * X7 + 1.24e4*max(0,0.345-X8) * max(0,X0-0.876) -
1.22e4*X7 * max(0,X0-1.31) + 1.21e4*max(0,X0-1.31) + 1.19e4*max(0,X1
0-2.84) * max(0,-0.148-X9) + 1.16e4*max(0,X8-2.01) * max(0,X9-0.741)
+ 1.06e4*max(0,0.345-X8) * max(0,X9-1.63) + 1.05e4*max(0,X9-0.296) *
max(0,X0-0.876) - 1.04e4*max(0,X8-2.01) * max(0,0.00720-X0) + 1.04e4
*max(0,X8-3.13) - 1.01e4*max(0,-0.427-X0) - 9774*max(0,X9-0.741) - 9
692*X7 * max(0,X8-2.01) - 9355*max(0,X10-1.49) * abs(X9) + 9277*max(
0,X10-1.49) * max(0,X9-0.296) + 8949*abs(X10) * max(0,X9-0.741) + 89
23*max(0,X9-1.63) + 8467*X7 * max(0,X10-2.17) + 8363*max(0,X0+0.427)
* max(0,-0.148-X9) + 8287*max(0,X0+0.427) * max(0,X8-2.01) + 8193*ma
x(0,-0.427-X0) * max(0,-0.592-X9) + 7181*abs(X9) * max(0,0.345-X8) -
7123*max(0,0.345-X8) + 6811*X7 * max(0,-0.148-X9) + 6794*X7 * max(0,
X2-2.70) - 6774*max(0,X2-1.58) * max(0,X0-0.00720) + 6697*max(0,0.34
5-X8) * max(0,X0-1.31) + 6167*max(0,X8-2.01) * max(0,-0.427-X0) + 61
26*max(0,X0-0.00720) * max(0,X9-1.63) + 5972*max(0,X10-0.151) + 5819
*abs(X10) * max(0,X9-0.296) - 5685*max(0,0.345-X8) * max(0,-0.148-X9
) + 5603*max(0,X2-1.58) + 5333*X11 - 5325*X11 * max(0,-0.592-X9) - 5
280*max(0,X8-2.01) * max(0,-0.148-X9) - 5230*X7 * abs(X9) - 5090*max
(0,X2-1.58) * max(0,X2-2.70) - 4924*max(0,0.345-X8) * X11 - 4779*max
(0,-0.148-X9) * max(0,-0.427-X0) + 4532*max(0,X1-2.87) - 4338*max(0,
X8-3.13) * abs(X9) + 4247*abs(X9) * max(0,X0-0.876) - 4184*X11 * max
(0,X8-2.01) + 4160*max(0,X0+0.427) * max(0,X9-1.63) + 3998*X7 * max(
0,X9-0.296) - 3995*max(0,X10-1.49) * max(0,X10-2.17) + 3679*max(0,X9
-0.296) * max(0,X0-1.31) + 3646*X13^2 - 3504*abs(X9) * max(0,-0.427-
X0) + 3475*abs(X9) * X11 + 3377*max(0,X10-0.823) + 3367*X7 * max(0,X
0-0.00720) - 3353*max(0,X0-0.00720) * abs(X9) - 3286*X11 * max(0,X0-
0.876) + 3280*max(0,X9-0.741) * max(0,X0-0.876) + 3204*max(0,X0-0.00
720) - 3146*max(0,X2-2.14) + 3133*max(0,X10-2.84) * max(0,X0+0.427)
- 3133*max(0,X10-2.17) * X11 - 3062*max(0,X9-0.741) * max(0,X0-1.31)
+ 2943*max(0,X2-1.58) * max(0,0.00720-X0) - 2909*max(0,X2-1.58) * ma
x(0,X10-1.49) + 2859*max(0,X0+0.427) * max(0,X10-2.17) - 2739*abs(X1
0) - 2736*max(0,X9-1.19) - 2694*X9^2 - 2524*max(0,0.00720-X0) * max(
0,-0.427-X0) - 2506*max(0,X8-2.57) - 2470*max(0,X2-1.02) + 2401*max(
0,X0+0.427) * max(0,X0-0.876) - 2355*X7 * max(0,-0.427-X0) + 2355*ma
x(0,X0-0.00720) * max(0,0.345-X8) + 2349*max(0,-0.0932-X2) + 2339*X7
* max(0,X9-0.741) - 2300*max(0,X9-0.296) * X11 - 2286*max(0,0.902-X8
) - 2212*max(0,X10-1.49) * X7 - 2170*max(0,X1-2.08) + 2151*max(0,X1-
0.494) + 2130*max(0,X10-2.17) * max(0,X0-0.00720) + 2090*max(0,X2-1.

```
58) * abs(X10) - 2024*X10^2 - 1960*max(0,X2-1.58) * X7 - 1886*max(0,
X1-1.29) + 1723*max(0,X9-0.296) * max(0,X9-1.63) - 1702*max(0,X0+0.4
27) * abs(X9) + 1624*max(0,X8-2.01) * max(0,-0.592-X9) - 1617*max(0,
-0.592-X9) + 1616*max(0,X2-1.58) * max(0,X8-2.01) - 1592*max(0,X10-1
.49) * max(0,X8-2.01) + 1592*X11 * max(0,-0.427-X0) - 1566*max(0,0.3
45-X8) * max(0,-0.427-X0) - 1561*abs(X9) * max(0,X0-1.31) + 1495*X7
* max(0,-0.592-X9) + 1478*max(0,X0-0.00720) * max(0,X0-0.876) - 1439
*X11 * max(0,-0.148-X9) + 1426*max(0,0.00720-X0) * max(0,-0.592-X9)
+ 1422*max(0,0.296-X9) + 1380*abs(X10) * max(0,X8-2.01) - 1374*max(0
,X2-1.58) * max(0,X9-0.296) - 1345*max(0,X2-1.58) * max(0,X10-2.17)
- 1342*max(0,X8-1.46) + 1280*max(0,X10-2.84) + 1243*max(0,-0.799-X13
) - 1240*max(0,-0.862-X0) + 1191*abs(X8) + 1160*X7 * max(0,X0-0.876)
- 1058*max(0,X0-0.00720) * X11 + 1050*max(0,-1.82-X12) - 1038*abs(X1
0) * max(0,0.00720-X0) + 1005*abs(X10) * max(0,X10-2.17) - 1002*abs(
X10) * max(0,X0+0.427) + 991*abs(X10) * X7 + 961*abs(X10) * max(0,X0
-0.00720) + 947*X11 * max(0,X9-0.741) - 934*max(0,X0+0.427) * max(0,
X9-0.741) - 916*abs(X10) * abs(X9) + 913*max(0,X9-0.296) - 866*max(0
,-1.09-X1) + 856*abs(X2) + 815*max(0,X0-0.442) - 742*abs(X11) + 732*
max(0,0.823-X10) - 709*abs(X1) - 671*X12 + 643*max(0,0.313-X11) + 63
1*max(0,-0.00400-X7) + 587*max(0,X10-2.17) + 570*abs(X0) + 566*max(0
,X0+0.427) * X11 + 532*max(0,0.749-X11) - 490*max(0,X0-0.00720) * ma
x(0,X9-0.741) - 472*max(0,-0.148-X9) * max(0,0.00720-X0) - 454*max(0
,X8-3.13) * max(0,X8-2.01) - 437*X4 + 433*max(0,0.494-X1) + 414*max(
0,-0.621-X6) - 401*abs(X5) - 374*abs(X10) * X11 + 374*max(0,X10-1.49
) * max(0,X0-0.00720) - 359*X5 - 286*max(0,1.46-X8) - 256*max(0,1.49
-X10) - 217*max(0,X1+0.297) + 215*abs(X12) + 214*max(0,-0.104-X5) -
187*X7 * X11 - 184*X11 * max(0,0.00720-X0) + 183*max(0,X10-1.49) * m
ax(0,X0+0.427) + 176*max(0,X10-1.49) * max(0,0.345-X8) - 172*X7 * ma
x(0,0.00720-X0) - 163*max(0,X9-0.296) * abs(X9) - 151*max(0,X8-3.13)
* X11 - 145*abs(X3) - 139*X6 + 126*max(0,X10-1.49) - 117*X0^2 + 99.3
*X7 + 83.5*max(0,-1.44-X12) - 81.6*abs(X9) * max(0,0.00720-X0) + 81.
1*max(0,-0.0819-X3) + 79.2*X8 - 74.9*X3 + 70.3*max(0,X9-0.296) * max
(0,0.345-X8) + 69.9*max(0,-0.157-X4) - 67.5*abs(X4) - 62.5*max(0,X12
+1.07) - 57.9*abs(X10) * max(0,0.345-X8) + 50.4*max(0,1.19-X11) + 48
.8*max(0,0.143-X4) + 42.0*max(0,0.206-X5) + 36.8*max(0,1.18-X3) + 36
.4*max(0,-0.0877-X6) + 24.2*max(0,1.62-X11) - 20.0*max(0,X5-0.826) +
15.1*X2^2 - 11.5*max(0,X10-1.49) * X11 - 10.6*max(0,0.741-X9) - 4.75
*max(0,X3-0.547) - 4.35*X11^2 + 2.93*max(0,0.744-X4) - 1.87*max(0,-0
.297-X1) - 1.82*X4^2 + 1.07*max(0,1.45-X5) + 0.661*max(0,0.233-X3) +
0.188*max(0,0.516-X5) + 0.0612*max(0,-0.354-X6) - 0.0479*max(0,X6+0.
354) - 0.00835*max(0,X11-0.313) + 0.00171*max(0,0.547-X3) - 8.93e-5*
max(0,X6+0.621) - 3.47e-6*max(0,X11-1.62) + 1.80e-7*max(0,0.179-X6)
```

Wow... whitebox but quite complicated!!!

We will now clean this equation into python code and create a new feature with it !

```
# CLEAN THE EQUATION INTO PYTHON CODE, TO EXECUTE IT AND CREATE A NEW FEATURE !
equation = equation.replace('log10', 'np.log10')
equation = equation.replace('max', 'np.maximum')
equation = equation.replace('^2', '**2')
for i in range(0, len(X_train_scaled.columns)):
    equation = equation.replace('X'+str(i), "X_train_scaled['"+ X_train_scaled.columns.tolist()[i] + "'].values")
for i in range (0, 10):
    equation = equation.replace('values'+str(i), 'values')
for i in range (0, 999):
    if len(str(i)) == 1:
        pos = '0' + str(i)
    else:
        pos = str(i)
    equation = equation.replace('values'+pos, 'values')
for i in range (0, 999):
    if len(str(i)) == 1:
        pos = '00' + str(i)
    else:
        pos = '0' + str(i)
    equation = equation.replace('values'+pos, 'values')
```

```
### APPLY THE SYMBOLIC REGRESSON EQUATION AND SCALE THE NEW FEATURE
scaler_ffx = StandardScaler()
X_train_scaled['FFX_SYMBOLIC_EQUATION'] = eval(equation)
X_train_scaled['FFX_SYMBOLIC_EQUATION'] = scaler_ffx.fit_transform(X_train_scaled[['FFX_SYMBOLIC_EQUATION']])
X_train['FFX_SYMBOLIC_EQUATION'] = X_train_scaled['FFX_SYMBOLIC_EQUATION']

X_test_scaled['FFX_SYMBOLIC_EQUATION'] = eval(equation.replace('X_train_scaled', 'X_test_scaled'))
X_test_scaled['FFX_SYMBOLIC_EQUATION'] = scaler_ffx.transform(X_test_scaled[['FFX_SYMBOLIC_EQUATION']])
X_test['FFX_SYMBOLIC_EQUATION'] = X_test_scaled['FFX_SYMBOLIC_EQUATION']
```

This is our final dataset; we just have added 8 new features form the original ones:

```
X_train_scaled.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 997 entries, 954 to 993
Data columns (total 15 columns):
 #   Column                Non-Null Count  Dtype  
---  ------                --------------  -----  
 0   age                   997 non-null    float64
 1   bmi                   997 non-null    float64
 2   children              997 non-null    float64
 3   region_northwest      997 non-null    float64
 4   region_southeast      997 non-null    float64
 5   region_southwest      997 non-null    float64
 6   sex_male              997 non-null    float64
 7   smoker_yes            997 non-null    float64
 8   bmi_smoker_yes        997 non-null    float64
 9   age2                  997 non-null    float64
 10  age_children          997 non-null    float64
 11  RULE_EXTRACT_1        997 non-null    float64
 12  RULE_EXTRACT_18       997 non-null    float64
 13  RULE_EXTRACT_8        997 non-null    float64
 14  FFX_SYMBOLIC_EQUATION  997 non-null    float64
dtypes: float64(15)
memory usage: 124.6 KB
```

To resume the feature engineering steps we just accomplished, we have created **8 NEW FEATURES**.

Each of them are whitebox and can be calculated with a linear formula (polynomials, rules or equation)

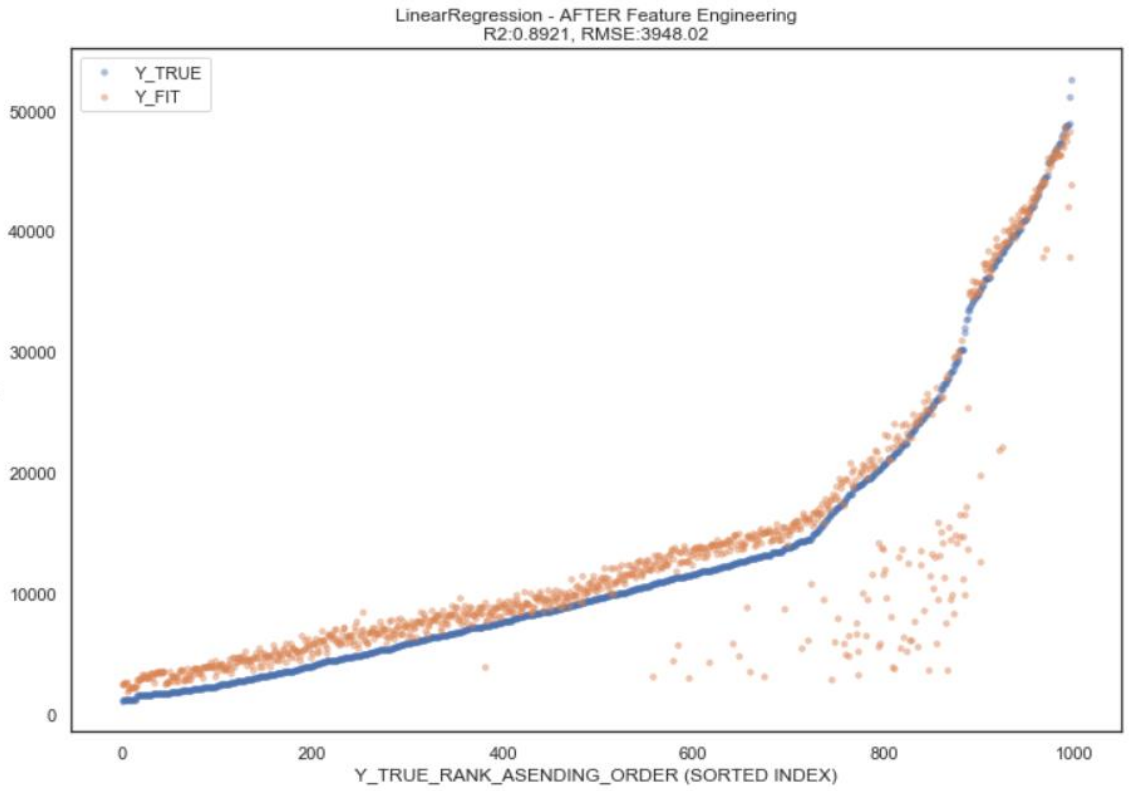
We are now ready to beat the notorious XGBoost!!!!

Linear Regression vs XGBoost AFTER feature engineering:

```
"""
###############################################
# 4-LINEAR REGRESSION MODEL WITHFEATURE ENGINEERING #
###############################################
"""

model_reg = LinearRegression()
model_reg.fit(X_train_scaled, y_train)
r2_linreg_after = model_reg.score(X_test_scaled, y_test)
rmse_linreg_after = np.sqrt(mean_squared_error(y_test, model_reg.predict(X_test_scaled)))
print('R2:' + str(round(r2_linreg_after, 4)) + ', RMSE:' + str(round(rmse_linreg_after, 2)))
```

```
R2:0.8921, RMSE:3948.02
```



Boom! Mission accomplished!

We just beat both regression scores of the notorious XGboost … with a simple Linear Regression model!

XGBoost BEFORE:

```
print('XGBoost R2 (Before feature engineering): ' + str(round(r2_xgb_before, 4)))
print('XGBoost RMSE (Before feature engineering): ' + str(round(rmse_xgb_before, 4)))
```

```
XGBoost R2 (Before feature engineering): 0.8869
XGBoost RMSE (Before feature engineering): 4043.6107
```

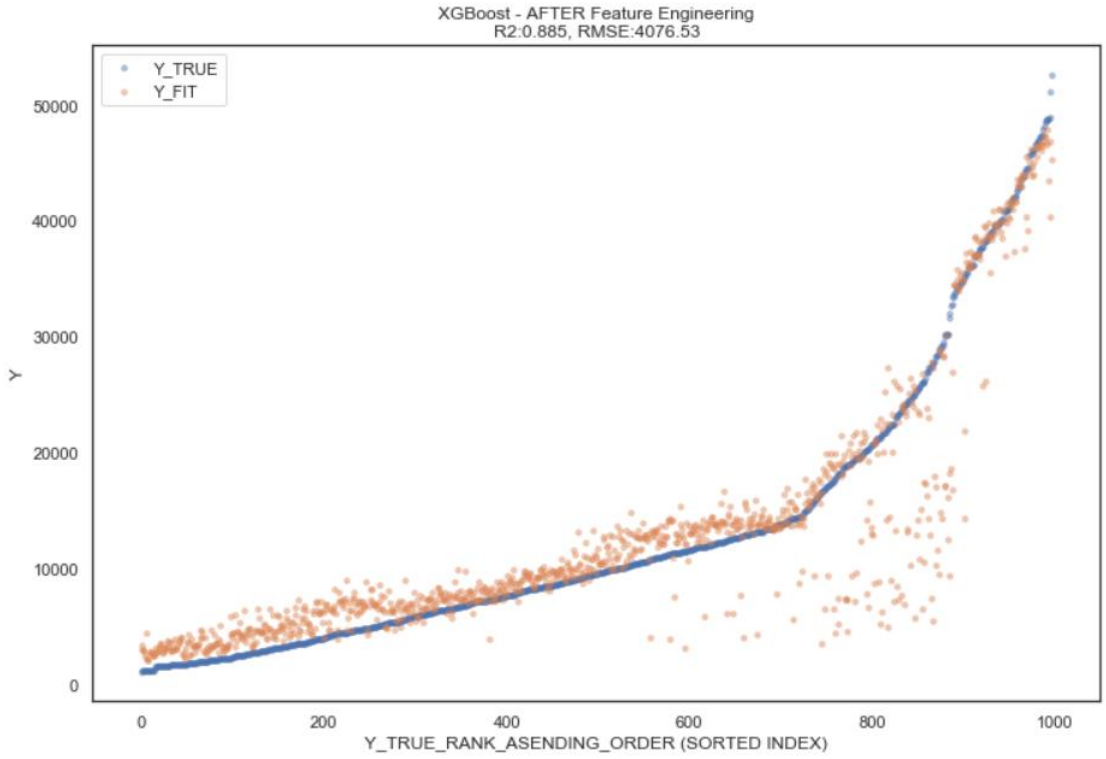
Linear Regression AFTER:

```
print('LinearRegression R2 (AFTER feature engineering): ' + str(round(r2_linreg_after, 4)))
print('LinearRegression RMSE (AFTER feature engineering): ' + str(round(rmse_linreg_after, 4)))
```

```
LinearRegression R2 (AFTER feature engineering): 0.8921
LinearRegression RMSE (AFTER feature engineering): 3948.016
```

Maybe the XGboost is now better than before with the new feature engineering process, let's try it! (With a new GridSearch of course!):

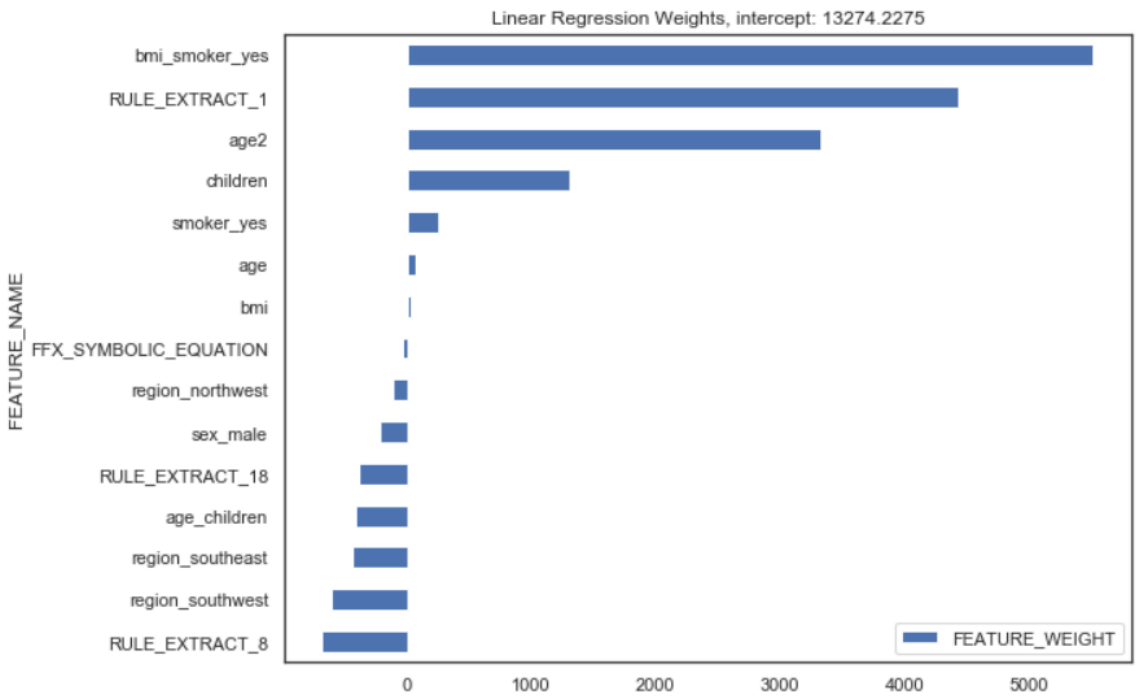
```
XGBRegressor(base_score=0.5, booster='gbtree', colsample_bylevel=1,
             colsample_bynode=1, colsample_bytree=0.6, gamma=0.3,
             importance_type='gain', learning_rate=0.1, max_delta_step=0,
             max_depth=3, min_child_weight=5, missing=None, n_estimators=50,
             n_jobs=1, nthread=None, objective='reg:linear', random_state=0,
             reg_alpha=0, reg_lambda=1, scale_pos_weight=1, seed=None,
             silent=None, subsample=0.8, verbosity=1)
R2:0.885, RMSE:4076.53
```



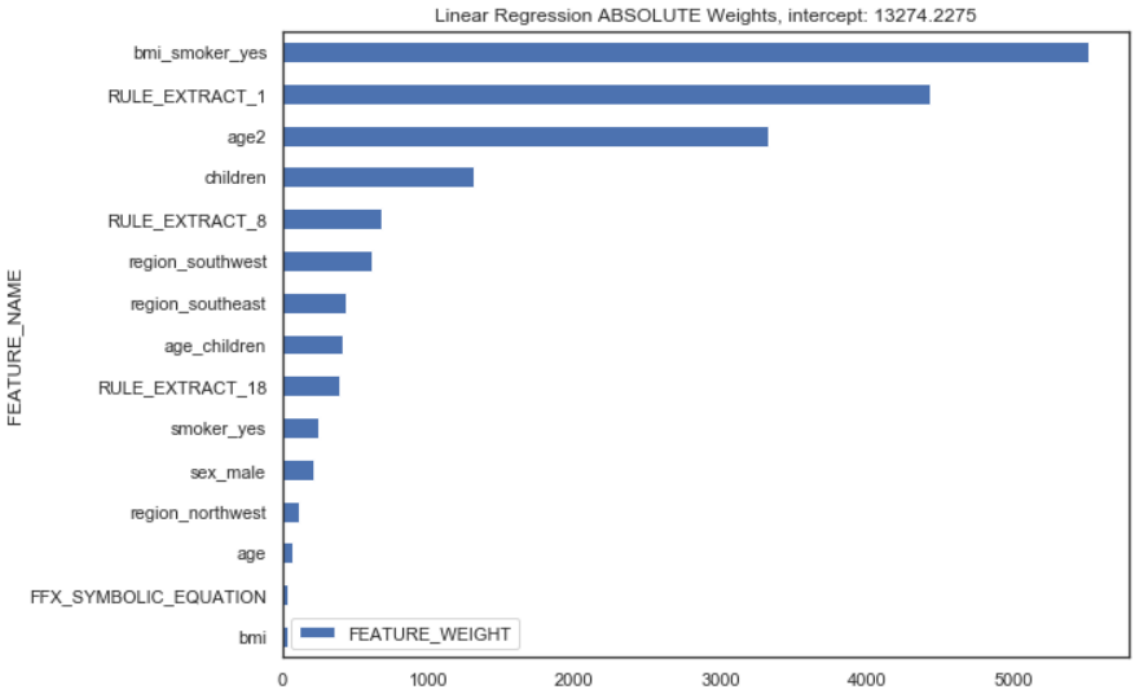
Boom! DOUBLE KILL - Mission accomplished twice!

Linear Regression model is still better! The new added features only helped the Linear Regression, not the XGBoost model

Let’s see the linear weights of the new Linear regression Model:



The absolute weights:




```
TOP 1:bmi_smoker_yes, absolute weight: 5521.3658
TOP 2:RULE_EXTRACT_1, absolute weight: 4433.0401
TOP 3:age2, absolute weight: 3325.6663
TOP 4:children, absolute weight: 1310.7841
```

```
TOP 5:RULE_EXTRACT_8, absolute weight: 679.8148
```

The FFX_SYMBOLIC_EQUATION feature have not added a lot of importance to the model...

Results:

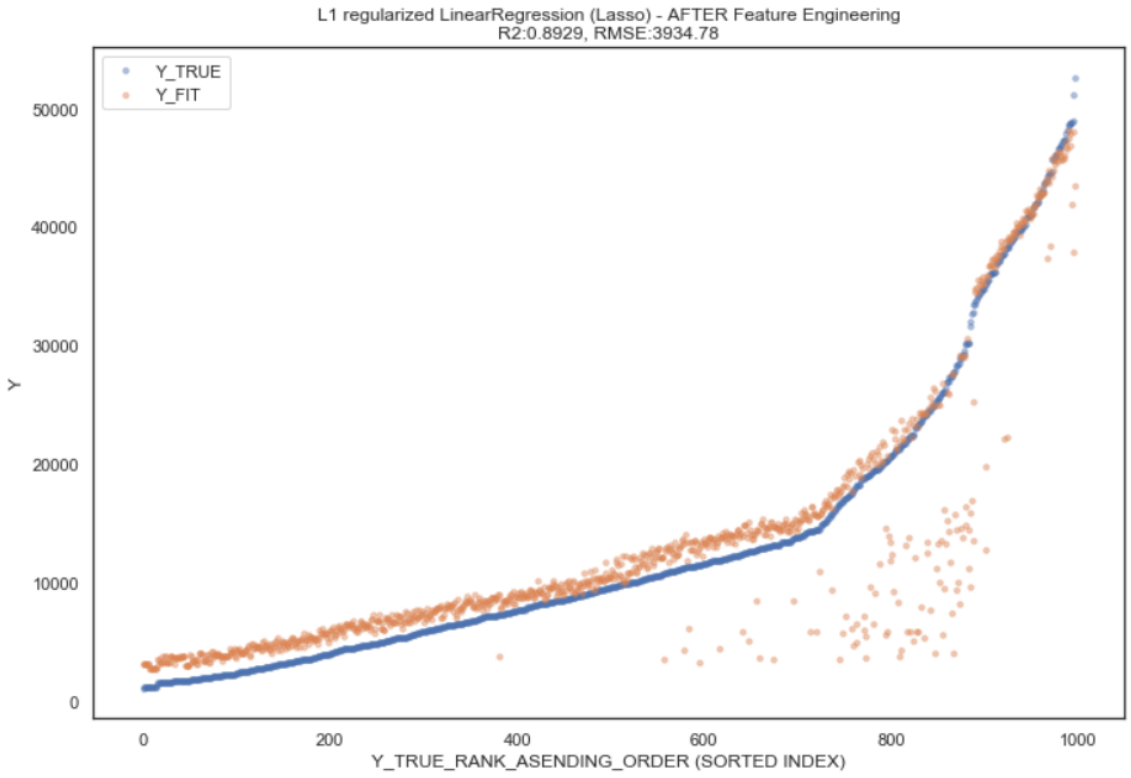
	METRIC	LINEAR_REGRESSION_BEFORE	XGBOOST_BEFORE	LINEAR_REGRESSION_AFTER	XGBOOST_AFTER
0	R2	0.780550	0.886855	0.892142	0.885006
1	RMSE	5631.448532	4043.610728	3948.015997	4076.528881

If we want to go further, we can still keep our model linear but add regularization to it. Let's see if a regularized linear model will be more powerful than a non-regularized linear regression (Lasso):

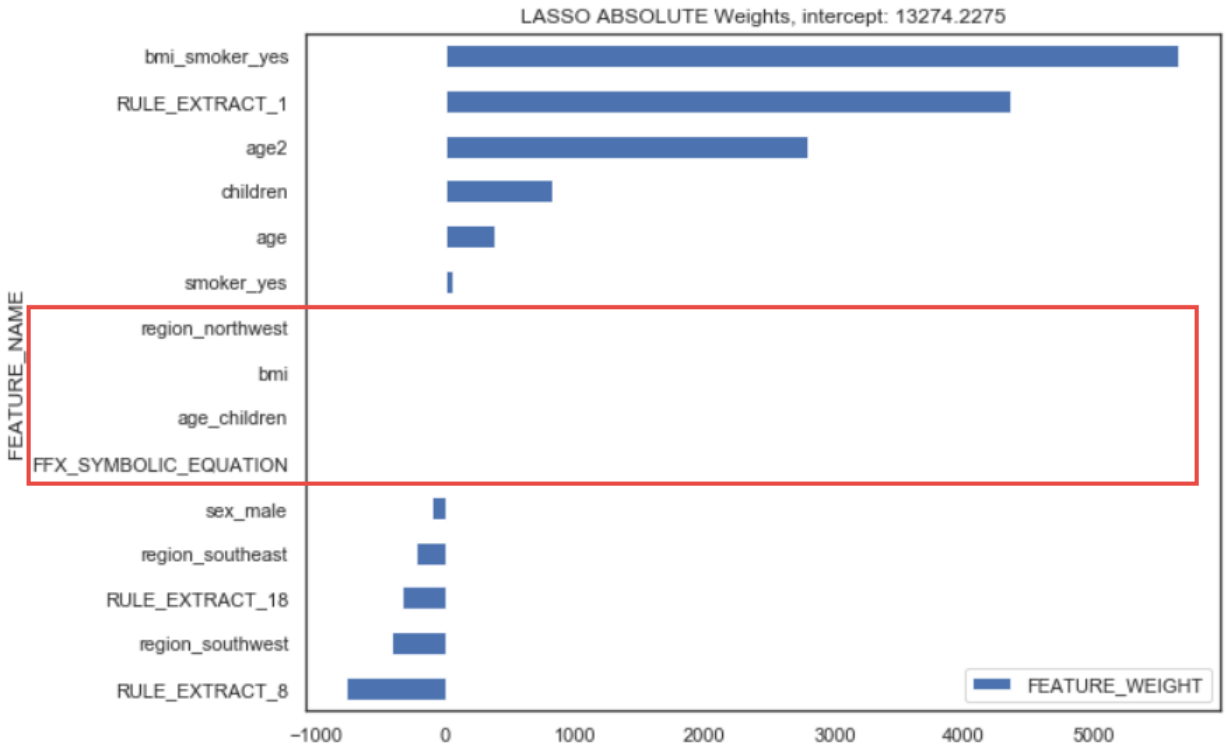
```
"""
#############################################
# L1 REGULARIZED LINEAR REGRESSION (LASSO) MODEL WITHFEATURE ENGINEERING #
#############################################
"""

model_reg = Lasso(alpha = 89)
model_reg.fit(X_train_scaled, y_train)
r2_lasso_after = model_reg.score(X_test_scaled, y_test)
rmse_lasso_after = np.sqrt(mean_squared_error(y_test, model_reg.predict(X_test_scaled)))
print('R2:' + str(round(r2_lasso_after, 4)) + ', RMSE:' + str(round(rmse_lasso_after, 2)))
```

```
R2:0.8929, RMSE:3934.78
```



YES!! The linear model is even better with L1 regularization !



With the linear lasso weights, we can see that the L1 regularization eliminated 4 features:

- Region_northwest
- Bmi
- Age_children
- FFX_Symbolic_equation

The model seems to be better without those features!!!

Conclusion

Sometimes, going forward with a more complex model or skipping the feature engineering step to hyper-parameters tuning is not always the best move to make.

Linear models are often UNDERRATED.

As a data science practitioner, I think we should pass a lot more time on feature engineering. It is easy to be lazy and go fast with a more complex and popular model like the XGboost.

Of course, I'm not crazy or blind; when we are in a big and complex data situation, linear models could fail very badly, even with a good feature engineering !, XGBoost / Deep neural networks are still gold nuggets in that kind of situations ! …And with **SHAPLEY VALUES** there is no black box anymore. We can get the feature importance from a Deep Neural Network if we want.

CREATIVITY: This should be the most important asset of any data science practitioners, not a PHD! There are much more to do then a simple ***SPLIT/FIT/PRED*** process... There are unlimited resources and tools out there, it is yours to be creative and play around with them!

What is nice about going linear is that now, I do not need a model anymore. I can just calculate the new features with the equations, apply the linear weights and that's it!

In addition, the RuleFit is a powerful underrated model with a lot of application. You should deep dive on that!

As I promised in my first part, I demonstrated again the power of feature engineering but this time, with a much more complicated dataset and by defeating XGBoost.

I hope this article have been interesting / useful to you!

Dave Cote, Data Scientist, M.Sc. BI