medical-costs-in-depth-regression-analysis

June 22, 2020

[1]:
```
# This Python 3 environment comes with many helpful analytics libraries␣
 ↪installed
# It is defined by the kaggle/python docker image: https://github.com/kaggle/
 ↪docker-python
# For example, here's several helpful packages to load in

import numpy as np # linear algebra
import pandas as pd # data processing, CSV file I/O (e.g. pd.read_csv)

# Input data files are available in the "../input/" directory.
# For example, running this (by clicking run or pressing Shift+Enter) will list␣
 ↪all files under the input directory

import os
for dirname, _, filenames in os.walk('/kaggle/input'):
    for filename in filenames:
        print(os.path.join(dirname, filename))

# Any results you write to the current directory are saved as output.
```

```
/kaggle/input/insurance/insurance.csv
```

[2]:
```
from pprint import pprint

# Pipeline
from sklearn.pipeline import Pipeline
from sklearn.compose import ColumnTransformer
from sklearn.impute import SimpleImputer

# Preprocessor
from sklearn.preprocessing import OneHotEncoder, LabelEncoder, StandardScaler
from sklearn.preprocessing import FunctionTransformer
from sklearn.preprocessing import PolynomialFeatures
from sklearn.preprocessing import PolynomialFeatures

# Model Selection
from sklearn.model_selection import cross_val_score
```

```
from sklearn.model_selection import cross_validate
from sklearn.model_selection import RandomizedSearchCV
from sklearn.model_selection import GridSearchCV
from sklearn.model_selection import cross_val_score, train_test_split

# Models
from sklearn.linear_model import LinearRegression, Ridge, Lasso
from xgboost import XGBRegressor
from sklearn.ensemble import RandomForestRegressor, AdaBoostRegressor,␣
↪GradientBoostingRegressor, VotingRegressor

# feature selection
from sklearn.feature_selection import SelectFromModel
from sklearn.feature_selection import RFECV

# metrics
from sklearn.metrics import r2_score
from sklearn.metrics import mean_squared_error
from sklearn.metrics import mean_absolute_error
from sklearn.metrics import make_scorer

# Statistics
from scipy import stats
from scipy.special import boxcox, inv_boxcox

# Seaborn
import seaborn as sns

# Yellowbrick Visualizations
from yellowbrick.model_selection import LearningCurve
from yellowbrick.regressor import PredictionError
from yellowbrick.regressor import CooksDistance
from yellowbrick.regressor import ResidualsPlot
from yellowbrick.model_selection import ValidationCurve

# Matplotlib
import matplotlib.pyplot as plt
%matplotlib inline

import warnings
warnings.filterwarnings('ignore')

SEED = 42
```

1 Helper Functions

1.1 Polynomial Functions

[3]:
```
def map_poly_feature_names(features, poly_features):
    """
    Given a list of the original feature names and new feature names generated␣
↪from a polynomial model, maps the new feature names to the
    original feature names.
    """

    d = {}

    # Maps the i-ith x to the i-th feature
    # e.g. 0, 1, and 2 is converted to x0, x1, and x2 respectively then mapped␣
↪such that d['x0'] = feature0
    for i, feature in enumerate(features):
        s = 'x'+ str(i)
        d[s] = feature

    # Maps x0, x1, ... to their respective feature names
    poly_feature_names = []
    for feature in poly_features:
        for key in d:
            if key in feature:
                feature = feature.replace(key, d[key])
        poly_feature_names.append(feature)
    return poly_feature_names
```

[4]:
```
def add_poly_features(df, degree, bias, interaction, target):
    """
    Given a dataframe, polynomial parameters, and the target column, fits and␣
↪transforms the numerical columns to have polynomial features.
    """

    isTargetNumerical = True if df[target].dtype in ['int64', 'float64'] else␣
↪False

    # If target is an int or float, we want to exclude it
    if isTargetNumerical:
        num_cols = list(df.select_dtypes(include=['int64','float64']).
↪drop(target, axis=1))
        cat_cols = list(df.select_dtypes(include=['object']))
    else:
        num_cols = list(df.select_dtypes(include=['int64','float64']))
        cat_cols = list(df.select_dtypes(include=['object']).drop(target,␣
↪axis=1))
```

```
    # Polynomial Model
    poly = PolynomialFeatures(degree, interaction_only=interaction,␣
↪include_bias = bias)

    # Fit and transform numerical columns
    poly_num_X = poly.fit_transform(df[num_cols])

    # Extract new feature names model
    poly_feature_names = poly.get_feature_names()

    # Map new feature names to the appropriate original feature names
    poly_feature_names = map_poly_feature_names(num_cols, poly_feature_names)

    # Combine new polynomial features with exisiting categorical columns
    poly_num_X = pd.DataFrame(poly_num_X, columns=poly_feature_names)
    poly_X = poly_num_X.join(df[cat_cols], how='inner')

    # Combine target with new DataFrame such that it's the right most column
    poly_X = poly_X.join(df[target], how='inner')

    return poly_X
```

1.2 Cross Validation Scoring Metrics

[5]:
```
# Custom scoring functions used for cross validation
def mae_scorer(y_true, y_pred):
    """ Returns the MAE score """
    y_true =  inv_boxcox(y_true, maxlog)
    y_pred =  inv_boxcox(y_pred, maxlog)
    return mean_absolute_error(y_true, y_pred)

def r2_scorer(y_true, y_pred):
    """ Returns the R2 score """
    y_true =  inv_boxcox(y_true, maxlog)
    y_pred =  inv_boxcox(y_pred, maxlog)
    return r2_score(y_true, y_pred)

def rmse_scorer(y_true, y_pred):
    """ Returns the RMSE score """
    y_true =  inv_boxcox(y_true, maxlog)
    y_pred =  inv_boxcox(y_pred, maxlog)
    return np.sqrt(mean_squared_error(y_true, y_pred))

def r2_adj_scorer(y_true, y_pred):
    """ Returns the adjusted R2 score """
    y_true =  inv_boxcox(y_true, maxlog)
```

```
    y_pred =  inv_boxcox(y_pred, maxlog)
    r2 = r2_score(y_true, y_pred)
    r2_adj = 1 - (((1 - r2) * (n - 1)) / ( n - k - 1))
    return r2_adj

# Custom Scorer
scoring = {'MAE': make_scorer(mae_scorer), 'RMSE': make_scorer(rmse_scorer),␣
↪'R2': make_scorer(r2_scorer), 'R2_Adjusted': make_scorer(r2_adj_scorer)}
```

2 Setup

[6]:
```
# Read in data
data = pd.read_csv('/kaggle/input/insurance/insurance.csv')
```

3 Exploratory Data Analysis

Visit my Exploratory Data Analysis to see insights into the dataset

4 Feature Engineering

From the EDA, we saw an interesting relationship between bmi groups for smokers and non-smokers which could serve as a useful predictor.

[7]:
```
data['bmi_smoker_risk'] = np.where((data.bmi < 26) & (data.smoker == 'no'),␣
↪'Healthy_No',
                           np.where((data.bmi < 26) & (data.smoker == 'yes'),␣
↪'Healthy_Yes',
                                    np.where((data.bmi > 25) & (data.bmi < 31)␣
↪& (data.smoker == 'no'), 'Overweight_No',
                                             np.where((data.bmi > 25) & (data.
↪bmi < 31) & (data.smoker == 'yes'), 'Overweight_Yes',
                                                      np.where((data.bmi > 30) &␣
↪(data.bmi < 41) & (data.smoker == 'no'), 'Obese_No',
                                                               np.where((data.bmi␣
↪> 30) & (data.bmi < 41) & (data.smoker == 'Yes'), 'Obese_Yes',
                                                                        np.
↪where((data.bmi > 40) & (data.smoker == 'no'), 'Morbid_No',
                                                                                 np.
↪where((data.bmi > 40) & (data.smoker == 'yes'), 'Morbid_Yes', 'None'))))))))
```

5 Data Preparation

[8]:
```
# Add_poly_features(df, degree, bias, interaction, target)
data = add_poly_features(data, 2, True, True, 'charges')

# Split data such that training and test have 80% and 20% respectively
split = int(data.shape[0] * .8)
train = data[:split]
test = data[split:]

# Select features and target for train and test sets
y_train = train['charges']
X_train = train.drop('charges', axis=1)

y_test = test['charges']
X_test = test.drop('charges', axis=1)

# Target is skewed so perform boxcox transformation
y_train, maxlog = stats.boxcox(y_train.values)

# Folds
folds = 10
```

[9]:
```
# Numerical features
numeric_features = train.select_dtypes(include=['int64','float64']).
 ↪drop('charges', axis=1).columns

# Retrieve a list of tuples with skewed feature names and their values
skewed_features_names = train[numeric_features].skew().index
skewed_features_values = train[numeric_features].skew()
skewed_feature_names_values = list(zip(skewed_features_names,␣
 ↪skewed_features_values))

# Numeric features with skew > .9
log_features = [feature for (feature, skew) in skewed_feature_names_values if␣
 ↪skew > .9]

# Remaing numeric features
scale_features = [name for name in numeric_features if name not in log_features]

# Categorical features
categorical_features = train.select_dtypes(include=['object']).columns
```

5.1 Pipelines

[10]:
```
# Pipeline to transform skewed data by imputation and scaling
numeric_scale_transformer = Pipeline(steps=[
    ('imputer', SimpleImputer(strategy='median')),
    ('scaler', StandardScaler())])

# Pipeline to transform skewed data by imputation, log + 1 transformation, and␣
 ↪scaling
numeric_log_scale_transformer = Pipeline(steps=[
    ('imputer', SimpleImputer(strategy='median')),
    ('log', FunctionTransformer(np.log1p, validate=False)),
    ('scaler', StandardScaler())])

# Pipeline to transform categorical data by imputation and one hot encoding
categorical_transformer = Pipeline(steps=[
    ('imputer', SimpleImputer(strategy='constant', fill_value='missing')),
    ('onehot', OneHotEncoder(handle_unknown='error', drop='first'))])

# Pipeline to preprocess data by calling other pipelines
preprocessor = ColumnTransformer(
    transformers=[
        ('num', numeric_scale_transformer, scale_features),
        ('log', numeric_log_scale_transformer, log_features),
        ('cat', categorical_transformer, categorical_features)])
```

6 Model Selection

[11]:
```
"""
R2 parameters: We need n, the number of observations in each sample, and k, the␣
 ↪number of predictors. To get these values, we must preprocess out data.

    n: For cross validation, I must consider the size of samples for each fold.

    k: The number of predictors. The only thing to consider is that since I␣
 ↪have a polynomial model, I also have a constant term. Therefore I subtract
       1 from the number of predictors
"""
n, k =  Pipeline(steps=[('preprocessor', preprocessor)]).fit(X_train, y_train).
 ↪transform(X_train).shape
n *= (1 - (1/folds))
n = int(n)
k -= 1

# Regression Classifiers
lr = LinearRegression()
```

```
ridge = Ridge(random_state = SEED)
lasso = Lasso(random_state = SEED)
rf = RandomForestRegressor(random_state = SEED)
xgb = XGBRegressor(objective ='reg:squarederror', random_state = SEED)
ada = AdaBoostRegressor(random_state = SEED)
gb = GradientBoostingRegressor(random_state = SEED)
vr = VotingRegressor([('lr', lr), ('ridge', ridge),('lasso',lasso),('rf',␣
↪rf),('xgb', xgb),('ada', ada),('grad',gb)])

classifiers = [
    ('LinearRegression', lr),
    ('Ridge', ridge),
    ('XGBRegressor', xgb),
    ('RandomForestRegressor', rf),
    ('AdaBoostRegressor', ada),
    ('GradientBoostingRegressor', gb),
    ('VotingRegressor', vr)
]

clf_names = []
clf_scores = []

# Calculate RMSE and R Squared for each classifier sorted by RMSE
for clf_name, clf_model in classifiers:

    # Append classifer name
    clf_names.append(clf_name)

    # Perform cross validation scoring
    pipe = Pipeline(steps=[('preprocessor', preprocessor),
                           ('model', clf_model)])

    cv_results = cross_validate(pipe, X_train, y_train, cv=folds,␣
↪scoring=scoring)

    mae = cv_results['test_MAE'].mean()
    rmse = cv_results['test_RMSE'].mean()
    r2 = cv_results['test_R2'].mean()
    r2_adj = cv_results['test_R2_Adjusted'].mean()
    clf_scores.append([mae, rmse, r2, r2_adj])

# DataFrame to display classifiers and their respective RMSE and score metric
pd.DataFrame(clf_scores, columns = ['MAE', 'RMSE','R2', 'R2 Adj'],␣
↪index=clf_names).sort_values('R2 Adj', ascending=False)
```

[11]:

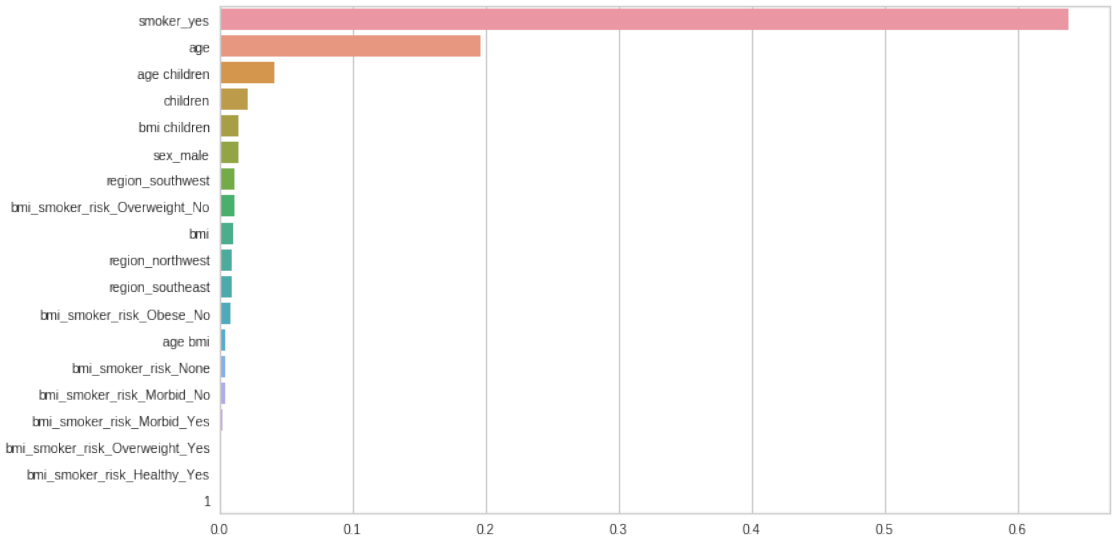
	MAE	RMSE	R2	R2 Adj
XGBRegressor	2077.007724	4448.851805	0.849404	0.846533

```
GradientBoostingRegressor  2111.458231  4508.354747  0.845268  0.842317
AdaBoostRegressor          3064.412939  4706.360742  0.834419  0.831261
RandomForestRegressor      2310.546982  4752.606072  0.829771  0.826525
VotingRegressor            2844.753080  5305.220062  0.793583  0.789647
Ridge                      3780.692738  7813.592014  0.557511  0.549074
LinearRegression           3814.812553  7913.341631  0.546608  0.537963
```

6.1 Feature Importance

[12]:
```
# XGBRegressor pipeline
clf_pipe = Pipeline(steps=[('preprocessor', preprocessor),
                           ('model', xgb)])

# Find the feature importance generated by the model of each fold
feature_importance_folds = []
feature_importance_cv = cross_validate(clf_pipe, X_train, y_train, cv=folds,␣
↪scoring=scoring, return_estimator = True)
for idx, estimator in enumerate(feature_importance_cv['estimator']):
    feature_importance_folds.append(list(estimator['model'].
↪feature_importances_))

# Extact feature names generated from OneHotEncoding
OH_categorical_features = clf_pipe.fit(X_train, y_train).
↪named_steps['preprocessor'].transformers_[2][1].named_steps['onehot'].
↪get_feature_names(categorical_features)

# Concatenate numeric feature names with the feature names genereated from␣
↪OneHotEncoding
feature_names = np.concatenate([numeric_features, OH_categorical_features])

# Plot feature importances
feature_ranking = pd.DataFrame(np.array(feature_importance_folds).T, index =␣
↪feature_names).mean(axis=1).sort_values(ascending=False)
plt.figure(figsize=(12,7))
sns.barplot(x=feature_ranking.values, y=feature_ranking.index)
```

[12]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x7fb48813bef0>
```



6.2 Feature Selection

[13]:
```
# Preprocess training set for feature selection
transformed_X_train = Pipeline(steps=[('preprocessor', preprocessor)]).
 ↪transform(X_train)
transformed_X_train = pd.DataFrame(transformed_X_train, columns=feature_names)

# Preprocess test set for feature selection
transformed_X_test = Pipeline(steps=[('preprocessor', preprocessor)]).
 ↪transform(X_test)
transformed_X_test = pd.DataFrame(transformed_X_test, columns=feature_names)

# Perform Recursive Feature Elimination Cross Validation
estimator = clf_pipe.named_steps['model']
selector = RFECV(estimator, scoring=make_scorer(mae_scorer))
selector = selector.fit(transformed_X_train, y_train)

# Extract features in order of importance ranking
clf_feature_rankings = list(zip(selector.ranking_, feature_names))
clf_feature_rankings.sort(key=lambda x: x[0])
clf_feature_rankings = [x for (_, x) in clf_feature_rankings]
```

[14]:
```
# Calculate scores for n features where features are added from most important␣
 ↪to least important
feature_scores = []
for i, _ in enumerate(clf_feature_rankings):
    features = clf_feature_rankings[:i+1]
```

```
    cv_results = cross_validate(xgb, transformed_X_train[features], y_train,␣
 ↪cv=folds, scoring=scoring)
    mae = cv_results['test_MAE'].mean()
    rmse = cv_results['test_RMSE'].mean()
    r2 = cv_results['test_R2'].mean()
    r2_adj = cv_results['test_R2_Adjusted'].mean()
    feature_scores.append([mae, rmse, r2, r2_adj])
feature_scores = pd.DataFrame(feature_scores, columns=['MAE','RMSE','R2','R2␣
 ↪adj'])
feature_scores.set_index(feature_scores.index + 1)
```

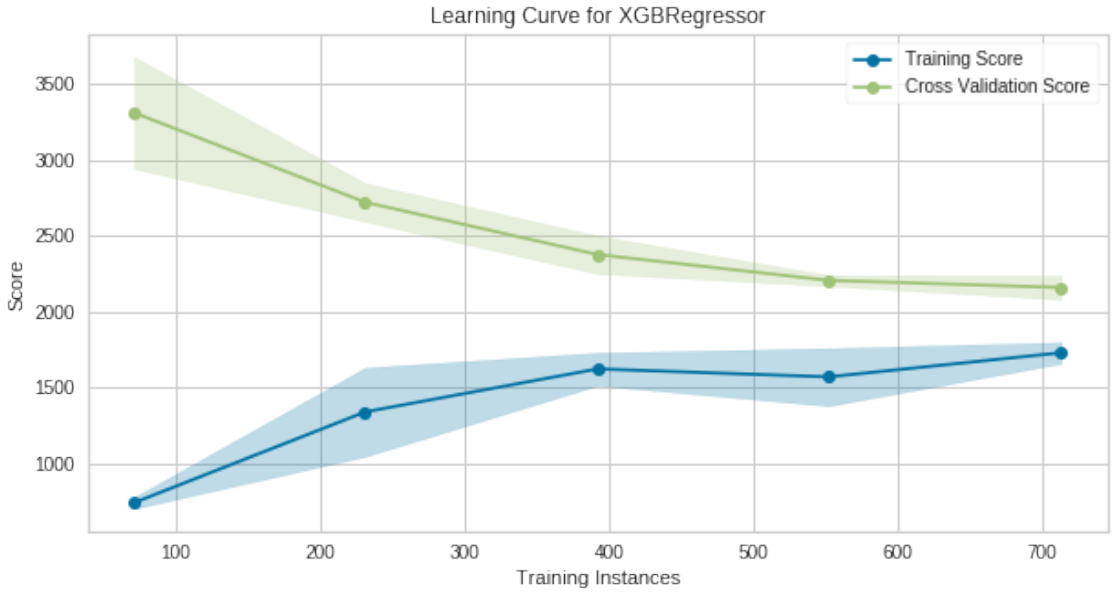
[14]:

	MAE	RMSE	R2	R2 adj
1	5740.745258	7729.824184	0.574785	0.566677
2	3868.603871	6602.272419	0.688329	0.682387
3	3785.303117	6565.487278	0.692477	0.686613
4	3789.567375	6609.550129	0.687937	0.681986
5	2965.000900	5381.820855	0.789489	0.785475
6	2938.513241	5346.576861	0.791845	0.787875
7	2945.671117	5344.956181	0.791731	0.787760
8	2924.349062	5328.341098	0.793896	0.789966
9	2102.363639	4502.413669	0.846165	0.843231
10	2104.349252	4484.242134	0.847877	0.844976
11	2088.069243	4456.491977	0.849070	0.846192
12	2097.331488	4457.714732	0.848904	0.846023
13	2100.084515	4466.070095	0.848568	0.845681
14	2074.185794	4449.023824	0.849635	0.846767
15	2078.316558	4447.753233	0.849783	0.846918
16	2078.316558	4447.753233	0.849783	0.846918
17	2073.841220	4444.315575	0.849654	0.846788
18	2073.841220	4444.315575	0.849654	0.846788
19	2073.841220	4444.315575	0.849654	0.846788

[15]:
```
# Set optimal feature count based on MAE score
optimal_feature_count = feature_scores.sort_values('MAE').index[0]
optimal_features = clf_feature_rankings[:optimal_feature_count]
print("Optimal number of features to minimize MAE on base model: {}".
 ↪format(optimal_feature_count))

# Filter features to only include features which give the optimal MAE score
feature_selection_X_train = transformed_X_train[optimal_features]
feature_selection_X_test = transformed_X_test[optimal_features]

# Re-calculate the number of predictors -- if 1 was kept, subtract 1
k = feature_selection_X_train.shape[1]
if '1' in feature_selection_X_train.columns:
    k -= 1
```

```
Optimal number of features to minimize MAE on base model: 18
```

[16]:
```
fig, ax = plt.subplots(figsize=(10,5))

untuned_estimator = estimator
visualizer = LearningCurve(untuned_estimator, scoring=make_scorer(mae_scorer),␣
↪ax=ax)
visualizer.fit(feature_selection_X_train, y_train)
visualizer.show()
```



[16]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x7fb488047c50>
```

As the number training instances increases, our model becomes more biased and less varianced. The learnining curve indicates that adding more data would definitely be beneficial.

7 Hyperparameter Tuning

7.1 Scoring Helper Functions

[17]:
```
def find_optimal_parameters(estimator, param_grid, random=False):
    """
    Given an estimater, parameter grid, and boolean indicating GridSearchCV or␣
↪RandomSearchCV, perform the respective search over the parameters.
    Print the optimal parameters founds and return the best fit estimator.
    """
    if random:
```

```
        grid_search = RandomizedSearchCV(estimator = estimator,␣
↪param_distributions = param_grid, n_iter = 100, cv = folds, n_jobs = -1,␣
↪scoring = make_scorer(mae_scorer, greater_is_better=False),random_state =␣
↪SEED)
        grid_search.fit(feature_selection_X_train, y_train)
        parameters = grid_search.best_params_
        for key in parameters:
            print('{}: {}'.format(key, parameters[key]))
        return grid_search.best_estimator_
    else:
        grid_search = GridSearchCV(estimator = estimator, param_grid =␣
↪param_grid, cv = folds, scoring = make_scorer(mae_scorer,␣
↪greater_is_better=False), n_jobs = -1)
        grid_search.fit(feature_selection_X_train, y_train)
        parameters = grid_search.best_params_
        for key in parameters:
            print('{}: {}'.format(key, parameters[key]))
        return grid_search.best_estimator_
```

[18]:
```
def score_model(model, model_name = 'Model'):
    """
    Given a model, performs cross validation scoring
    """
    clf_scores = []
    cv_results = cross_validate(model, feature_selection_X_train, y_train,␣
↪cv=folds, scoring=scoring)
    mae = cv_results['test_MAE'].mean()
    rmse = cv_results['test_RMSE'].mean()
    r2 = cv_results['test_R2'].mean()
    r2_adj = cv_results['test_R2_Adjusted'].mean()
    clf_scores.append([mae, rmse, r2, r2_adj])
    return pd.DataFrame(clf_scores, columns=['MAE','RMSE','R2','R2 adj'],␣
↪index=[model_name])
```

[19]:
```
def Validation_Curve_Visualization(model, parameter, values, X, y):
    fig, ax = plt.subplots(figsize=(10,5))

    viz = ValidationCurve(model, param_name=parameter, param_range=values,␣
↪cv=folds, scoring=make_scorer(mae_scorer))
    viz.fit(X, y)
    viz.show()
```

7.2 Parameters for XGBRegressor

[20]:
```
# Examine Hyperparameters
xgb_base = XGBRegressor(objective ='reg:squarederror', random_state = SEED)
pprint(xgb.get_params())
```

```
{'base_score': 0.5,
 'booster': 'gbtree',
 'colsample_bylevel': 1,
 'colsample_bynode': 1,
 'colsample_bytree': 1,
 'gamma': 0,
 'importance_type': 'gain',
 'learning_rate': 0.1,
 'max_delta_step': 0,
 'max_depth': 3,
 'min_child_weight': 1,
 'missing': None,
 'n_estimators': 100,
 'n_jobs': 1,
 'nthread': None,
 'objective': 'reg:squarederror',
 'random_state': 42,
 'reg_alpha': 0,
 'reg_lambda': 1,
 'scale_pos_weight': 1,
 'seed': None,
 'silent': None,
 'subsample': 1,
 'verbosity': 1}
```

7.3 Baseline Model

[21]:
```
# Score Baseline Model
score_model(xgb_base, 'Baseline Model')
```

[21]:

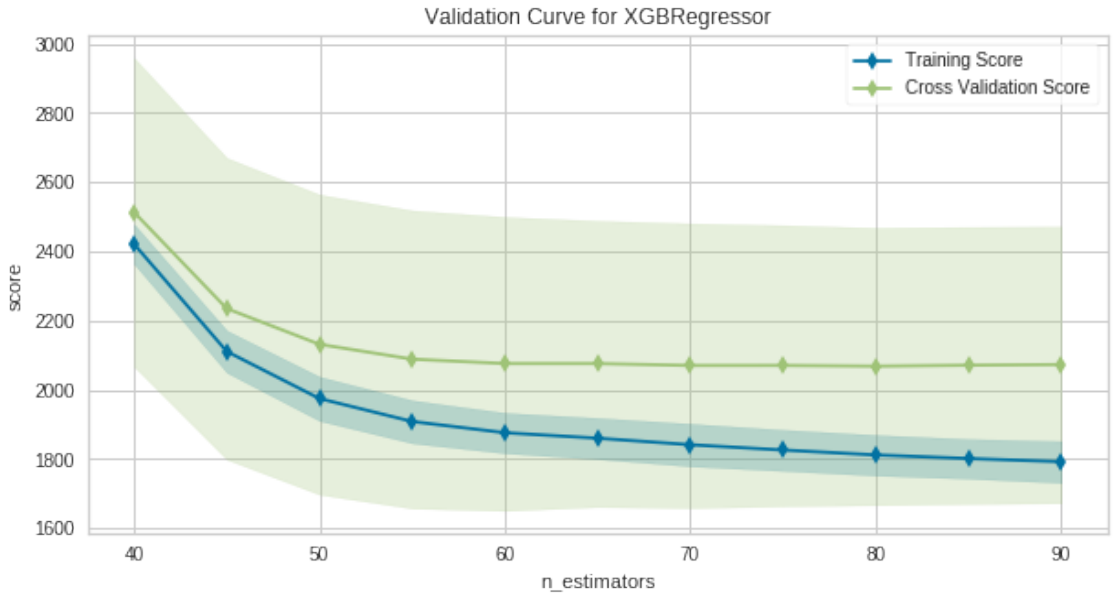
	MAE	RMSE	R2	R2 adj
Baseline Model	2073.84122	4444.315575	0.849654	0.846788

7.4 Tune Parameters

First find the best number of estimators for a fixed learning rate

[22]:
```
param_grid = {
    'n_estimators': [x for x in range(40, 95,5)]
}
```

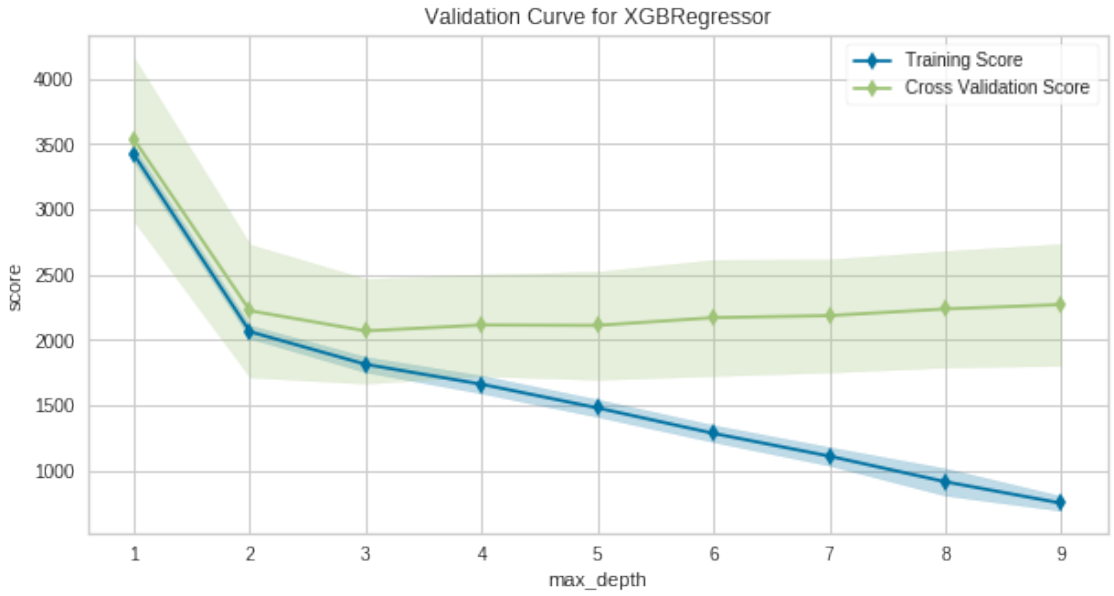
```
Validation_Curve_Visualization(xgb_base, 'n_estimators',␣
 ↪param_grid['n_estimators'],  feature_selection_X_train, y_train)
estimator = find_optimal_parameters(xgb_base, param_grid, False)
```




```
n_estimators: 80
```

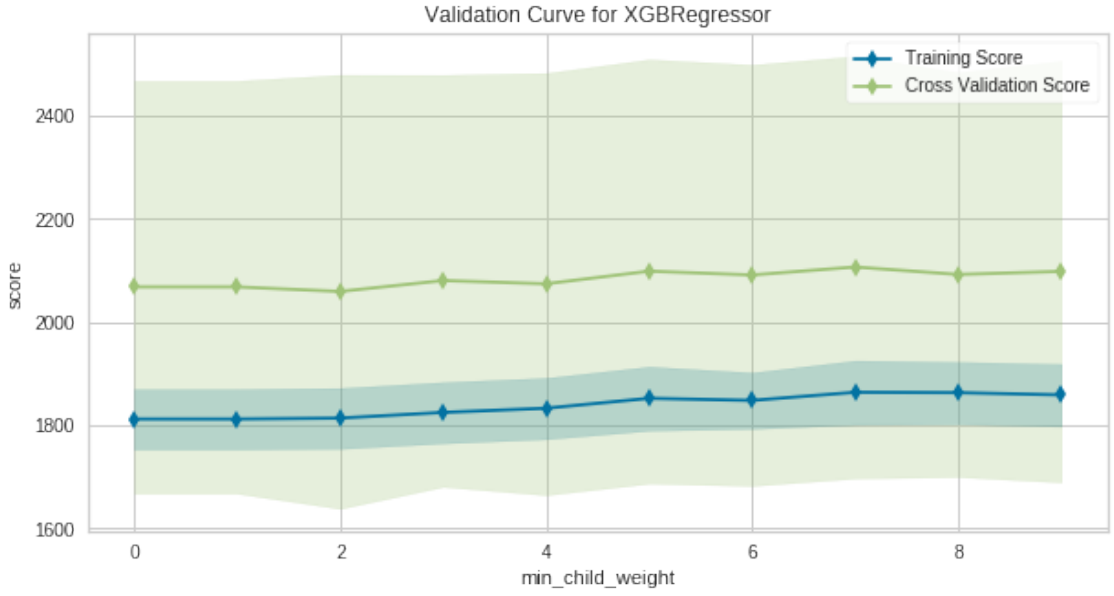
Tune tree-specific parameters for a fixed learning rate and number of trees * max_depth * min_child_weight * gamma, subsample * colsample_bytree

[23]:
```
param_grid = {
    'max_depth': [x for x in range(1,10,1)]
}

Validation_Curve_Visualization(estimator, 'max_depth', param_grid['max_depth'],␣
 ↪ feature_selection_X_train, y_train)
estimator = find_optimal_parameters(estimator, param_grid, False)
```



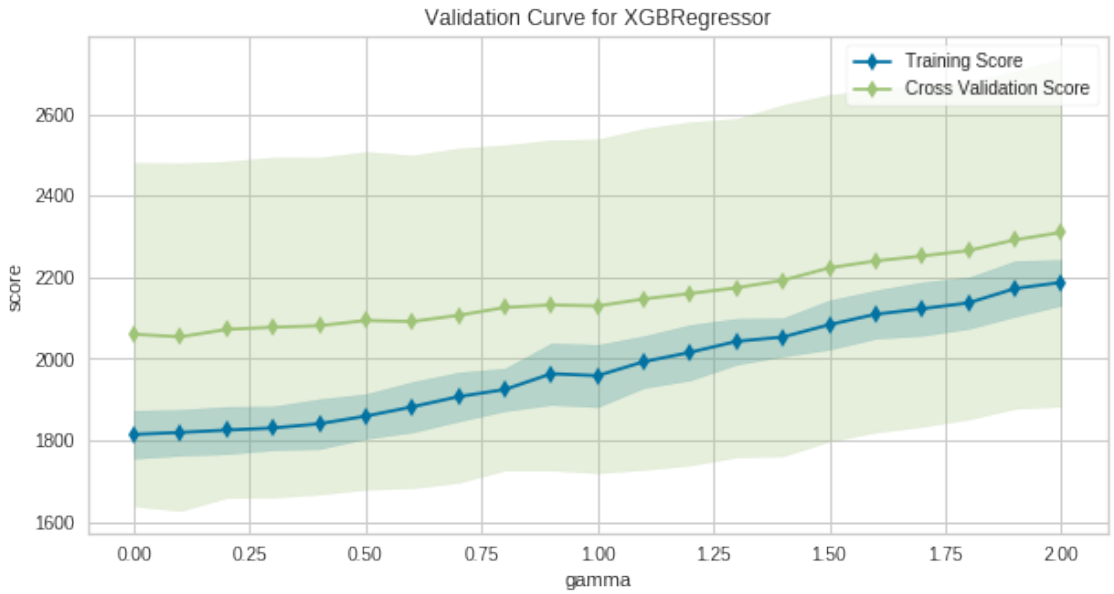
max_depth: 3

[24]:
```
param_grid = {
    'min_child_weight': [x for x in range(0,10,1)]
}

Validation_Curve_Visualization(estimator, 'min_child_weight',␣
↪param_grid['min_child_weight'],  feature_selection_X_train, y_train)
estimator = find_optimal_parameters(estimator, param_grid, False)
```



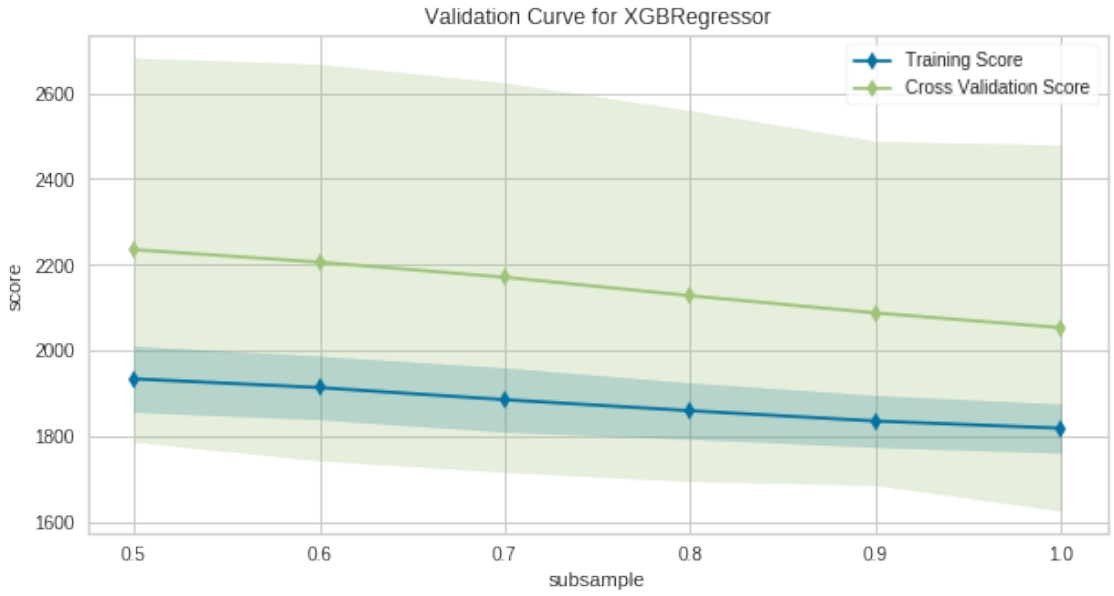
```
min_child_weight: 2
```

[25]:
```
param_grid = {
    'gamma': np.linspace(0,2,21)
}

Validation_Curve_Visualization(estimator, 'gamma', param_grid['gamma'], ␣
 ↪feature_selection_X_train, y_train)
estimator = find_optimal_parameters(estimator, param_grid, False)
```



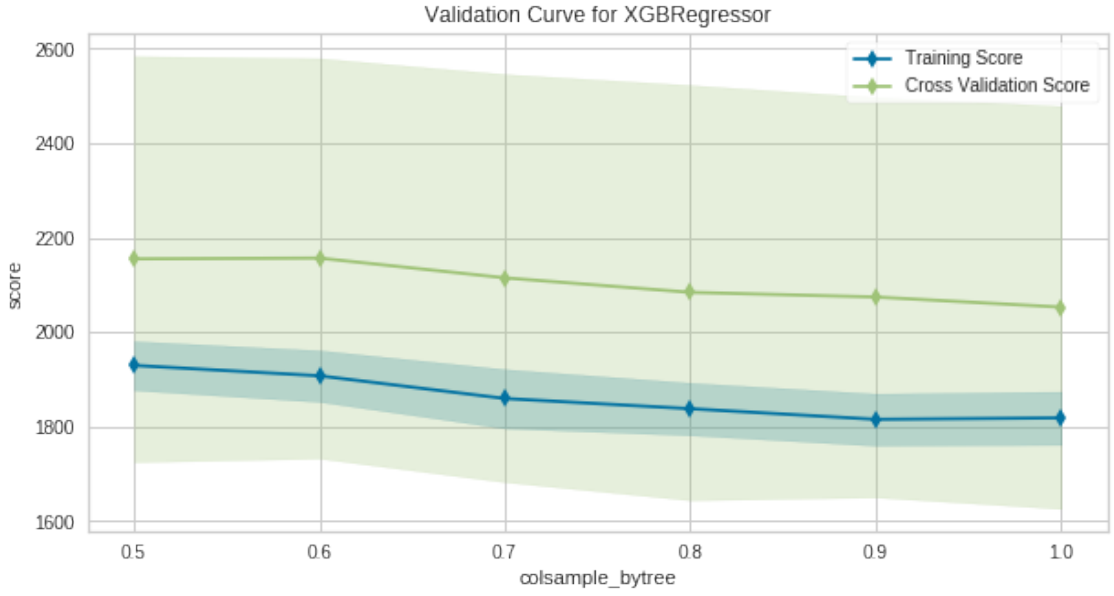

```
gamma: 0.1
```

[26]:
```
param_grid = {
    'subsample': np.linspace(0.5,1,6)
}

Validation_Curve_Visualization(estimator, 'subsample', param_grid['subsample'],␣
 ↪ feature_selection_X_train, y_train)
estimator = find_optimal_parameters(estimator, param_grid, False)
```



subsample: 1.0

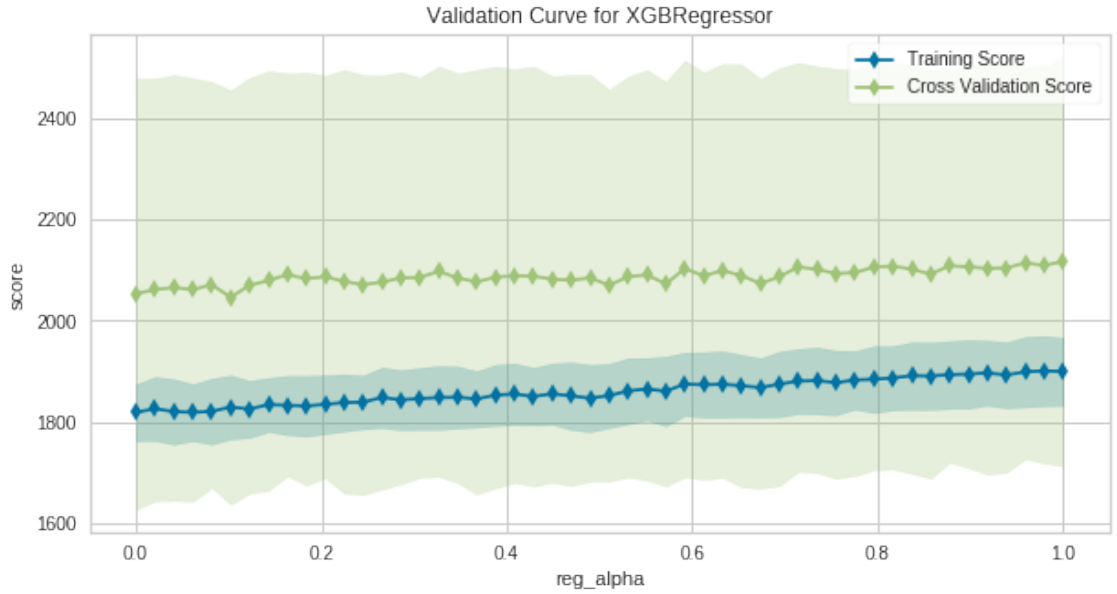
[27]:
```
param_grid = {
    'colsample_bytree': np.linspace(0.5,1,6)
}

Validation_Curve_Visualization(estimator, 'colsample_bytree',␣
 ↪param_grid['colsample_bytree'],  feature_selection_X_train, y_train)
estimator = find_optimal_parameters(estimator, param_grid, False)
```



```
colsample_bytree: 1.0
```

After tuning specific tree parameters, find the optimal regularization parameter

[28]:
```
param_grid = {
    'reg_alpha': np.linspace(0,1)
}

Validation_Curve_Visualization(estimator, 'reg_alpha', param_grid['reg_alpha'],␣
 ↪ feature_selection_X_train, y_train)
estimator = find_optimal_parameters(estimator, param_grid, False)
```




```
reg_alpha: 0.1020408163265306
```

Finally, perform a grid search lowering the learning rate (default of 0.1) and finding the number of estimators to minimize MAE with said learning_rate

[29]:
```
random_grid = {
    'learning_rate': np.linspace(0, .1),
    'n_estimators': [x for x in range(0,400,10)]
}

estimator = find_optimal_parameters(estimator, random_grid, True)
```

```
n_estimators: 130
learning_rate: 0.0673469387755102
```

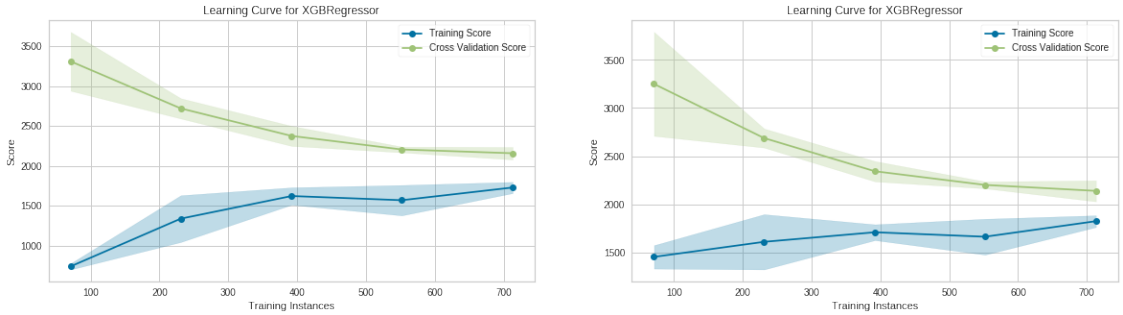
[30]:
```
# Optimized MAE Score
score_model(estimator)
```

[30]:

	MAE	RMSE	R2	R2 adj
Model	2053.608081	4432.638684	0.850738	0.847892

7.5 Is there enough data?

[31]:
```
fig, ax = plt.subplots(ncols=2, figsize=(20,5))

# Learning curve for base model
visualizer = LearningCurve(untuned_estimator, scoring=make_scorer(mae_scorer),␣
 ↪ax=ax[0])
visualizer.fit(feature_selection_X_train, y_train)
visualizer.finalize()

# Learning curve for tuned model
visualizer = LearningCurve(estimator, scoring=make_scorer(mae_scorer), ax=ax[1])
visualizer.fit(feature_selection_X_train, y_train)
visualizer.finalize()
visualizer.show()
```



[31]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x7fb482831320>
```

The model on the left indicates a base model with our selected features. The model on the right indicates our tuned model with our selected features. We can that while there is a little bit more bias, the scores have less variance now. Our tuned model still indicates that we could use some more data.

8 Model Predictions

[32]:
```
# Best Model
best_model = estimator
```

```
# Fit
best_model.fit(feature_selection_X_train, y_train)

# Predict
y_pred_transformed = best_model.predict(feature_selection_X_test)
y_pred_untransformed = inv_boxcox(y_pred_transformed, maxlog)

# Apply boxcox transformation to test
y_test_transformed =  boxcox(y_test, maxlog)

# Residuals
y_residuals = y_test_transformed - y_pred_transformed
```

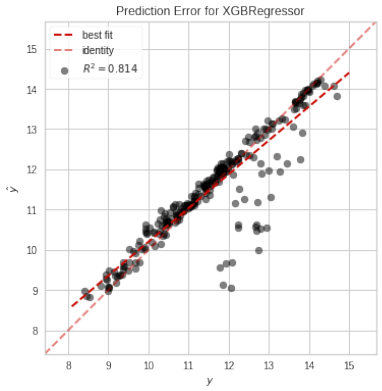
[33]:
```
# Score
mae_best = mean_absolute_error(y_test, y_pred_untransformed)
rmse_best = np.sqrt(mean_squared_error(y_test, y_pred_untransformed))
r2_best = r2_score(y_test, y_pred_untransformed)
r2_adjusted_best = 1 - (((1 - r2) * (n - 1)) / ( n - k - 1))
pd.DataFrame([[mae_best, rmse_best, r2_best, r2_adjusted_best]],␣
↪columns=['MAE','RMSE','R2','R2 Adj'], index=['Scores on Test Set '])
```

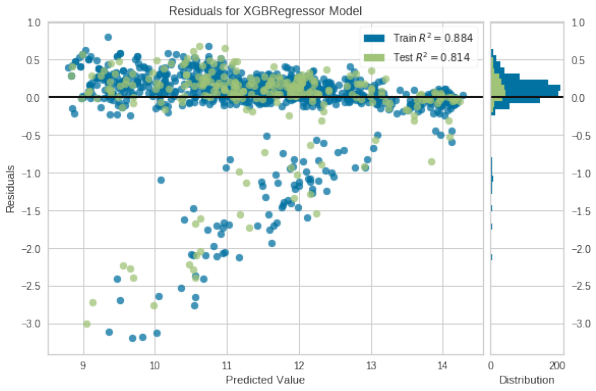
[33]:

	MAE	RMSE	R2	R2 Adj
Scores on Test Set	2227.969797	4582.803783	0.863559	0.846788

8.1 Predicted vs Residuals

[34]:
```
fig, ax = plt.subplots(ncols = 2, figsize=(20,6))

visual_grid = [
    PredictionError(best_model, line_color = 'r', point_color = 'black', alpha␣
↪= '.5', ax=ax[0]),
    ResidualsPlot(best_model, ax=ax[1])
]

for visual in visual_grid:
    visual.fit(feature_selection_X_train, y_train)
    visual.score(feature_selection_X_test, y_test_transformed)
    visual.finalize()
visual.show()
```



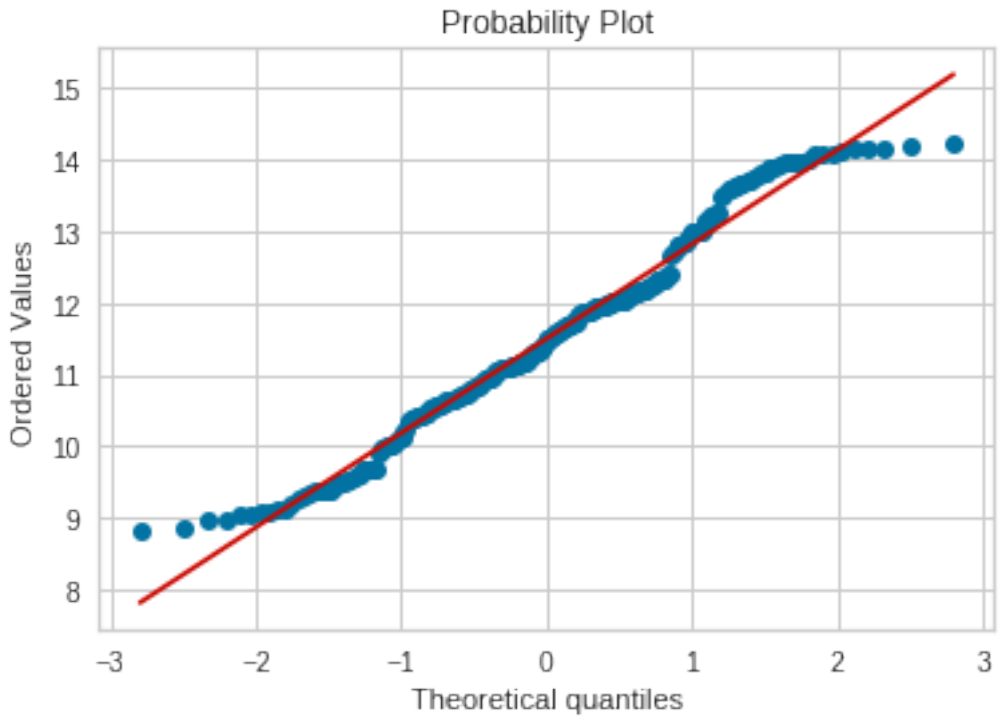


[34]: `<matplotlib.axes._subplots.AxesSubplot at 0x7fb483b877f0>`

Ideally, our plot shouldn't show any pattern. However, there seems to be a trend where we predict some values too high. A reasoning for this could be attributed to the fact that we don't have a normal distribution of charges. That is, we have a subset of charges that are much higher than the average charges with much lower frequency. Moreover, we might not have enough features to to help distinguish what causes these high prices from those who don't have such high charges. More data and features can help remedy these problems and improve our model.

8.2 QQ-Plot

[35]:
```
_ = stats.probplot(y_pred_transformed, plot=plt)
```

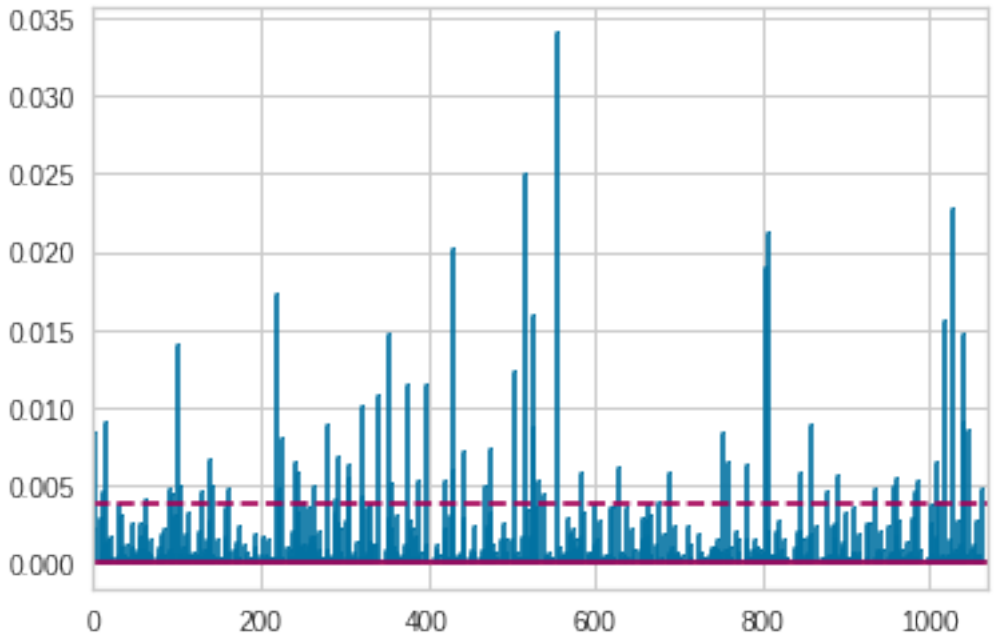


The normal distribution of errors is roughly a straight line where the ends seem to trail off. An ideal distribution follows a stright line strictly so this probability plot is promsiing.

8.3 Cook's Distance

[36]:
```
# Plot
visualizer = CooksDistance()
visualizer.fit(feature_selection_X_train, y_train)
```

[36]:
```
CooksDistance(ax=<matplotlib.axes._subplots.AxesSubplot object at
0x7fb480665908>,
              draw_threshold=True, linefmt='C0-', markerfmt=',')
```



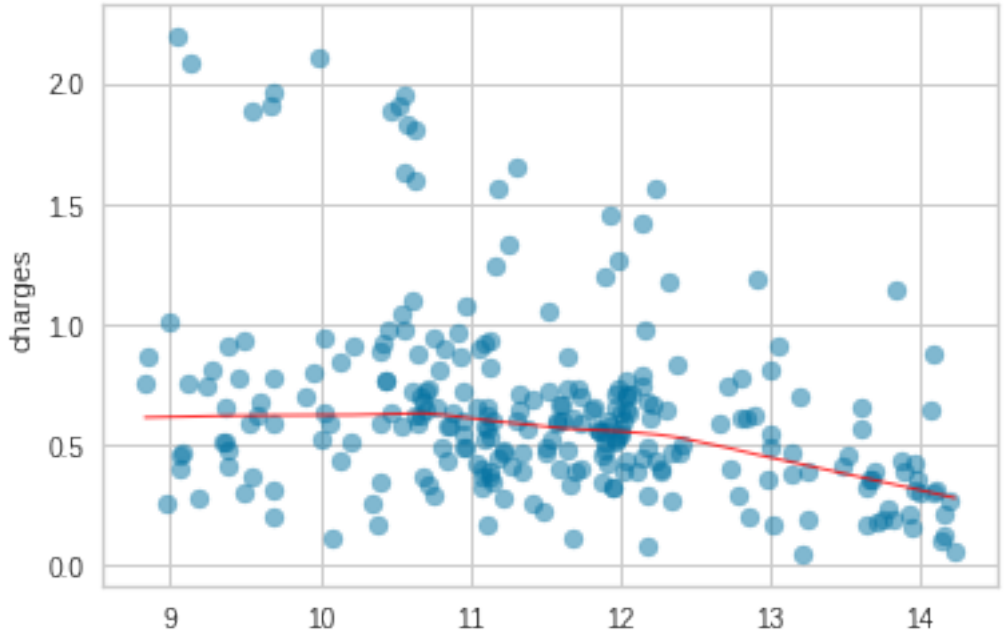
Cook's distance shows that we have quite a few infuential cases. It's important to note that influential cases are not usually a problem when their removal from the dataset would leave the parameter estimates essentially unchanged therefore the ones we worry about are those whose presence really does change the results.

8.4 Scale Location Plot

[37]:
```
# The square root of the absolute standarized residuals
sqrt_abs_standardized_residuals = np.sqrt(abs((y_residuals - y_residuals.
↪mean())/y_residuals.std()))
```

```
# Plot
plt.scatter(y_pred_transformed, sqrt_abs_standardized_residuals, alpha=.5)
sns.regplot(y_pred_transformed, sqrt_abs_standardized_residuals,
            scatter=False,
            ci=False,
            lowess=True,
            line_kws={'color': 'red', 'lw': 1, 'alpha': 0.8})
```

[37]: <matplotlib.axes._subplots.AxesSubplot at 0x7fb4724f19b0>



This plot is useful to determine heteroskedasticity which is present when the size of the error term differs across values of an independent variable. Ideally, we wouldn't want this plot to show any pattern and for it have a relatively straight line. The information in this plot can be corroborated with the information in the **Predicted vs Residuals** plot above. That is, we have a few large values for charges and not enough features to describe the data accurately enough higher predictions for charges with lesser value.