Analysis of Quantitative Regression QSAR Models for Predicting Chemical Toxicity

New regulations requiring acute toxicity testing of chemicals towards fish and supports the use of QSAR models for compliance. Pre-curated acute toxicity data for 908 chemicals was analyzed, processed, and modeled in order to develop an accurate and scalable model. Reducing the data via Principal Component Analysis and UMAP showed potential correlations between chemical descriptors and toxicity. These correlations were more pronounced with UMAP, suggesting non-linear interaction between descriptors and toxicity. Multiple regression, regularization, gradient descent, random forest, neural networks, and k nearest neighbor were applied on a training set of 726 chemicals and validated on 182 chemicals in order to predict LC_{50} 96 hours for the fathead minnow. The k nearest neighbor model had the best overall performance on the validation set, preserving nonlinear trends while preventing overfitting. The k nearest neighbor model scales well when adding new instances, improving the accuracy of each cluster. Suggestions were made by the researcher to further explore toxicity, by converting toxicity into a categorical feature using a cutoff value for LC_{50} in order to classify whether a chemical is safe.

INTRODUCTION

With the introduction and enforcement of REACH regulations in 2007 came a surge of interest in computational studies on chemical toxicity [1]. With a requirement of proving products are safe for humans and the environment before introducing them to the market and a goal of ending unnecessary animal testing, REACH promoted alternative testing methods. Quantitative structure–activity relationships (QSAR) emerged as the go-to approach regarding chemical testing. QSAR aims to find functional relationships in chemical structure and their properties.

As part of aquatic testing, REACH requires evaluation of short-term toxicity towards fish on goods imported or manufactured more than 10 tons per year [1]. QSAR modeling provides an economic cost and animal welfare benefit. Past studies utilizing QSAR point to two methods of predicting toxicity—some classifying chemicals based on their mode of action and others by estimating a quantitative parameter [3-5]. The second method is applied at various scales. Some seek to apply the quantitative response techniques to chemicals with similar modes of action [6-8]. The success of this method depends on the ability to identify each chemical's mode of action before applying it to a model, a computationally expensive task. The other scale the method is applied to is a global scale, which takes a large set of chemicals and models them. The heterogenous nature of these chemicals can add noise to the models, decreasing accuracy.

This study focuses utilizing different modelling approaches to model a quantitative response variable (LC_{50}) at a global scale. This aims to address current problems in the literature: poor scaling, small domains of applicability. Past studies have attempted to address the issue of applicability by building complex neural networks [21-25], but due to their initial computational cost and complexity, they scale poorly with small datasets and are prone to overfitting. On the contrary, linear regression algorithms have much less cost and better scaling but tend to underfit the data as they do not consider nonlinear trends [9-20]. Models in this work will be based on six molecular descriptors in order to predict the toxicity.

MATERIALS AND METHODOLOGY

Data

Data is from the QSAR fish toxicity Data Set located within the UCI Machine Learning Repository [2]. The data and is comprised of 928 chemicals, each containing 6 molecular descriptors: MLOGP (molecular properties), CIC0 (information indices), GATS1i (2D autocorrelations), NdssC (atom-type counts), NdsCH (atom-type counts), SM1_Dz(Z) (2D matrix-based descriptors) and a quantitative response variable LC_{50} which is the concentration of a chemical that causes death in 50% of test fish over a duration of 96 hours.

Environment

The analysis and modeling of the data was performed in an anaconda environment containing Python 3.9.7 and the following dependencies: pandas, numpy, scikit-learn, statsmodels, matplotlib, seaborn, tensorflow, and umap [26-33]. Random seed of 101 was set for all stochastic operations in order to maintain consistency between runs.

Analysis

The dataset was loaded via the pandas module and inspected for potential outliers and missing values. A pairplot of the data was produced and any interesting correlations were plotted in higher resolution. Pairplots using the discreet variables NdssC and NdssCH were used to inspect potential collinearity between them and the continuous variables. Correlation matrix was produced in order to investigate potential multicollinearity between the descriptors. Data was then z-score normalized and split into a training set, with 80% of the data, and a validation set with the other 20%. The 6 descriptors were then transformed via Principle Component Analysis (PCA) and UMAP fitted against l1 target metric into 2 dimensions, allowing for visualization of trends in the data.

Modelling

Univariate regression models were developed for each feature in order to provide a baseline for a multivariate model. The first multivariate model had each feature without any consideration for interaction. It was then reduced removing non-significant variables and ANOVA was performed to confirm there was no statistical difference between the models. Then

to test interaction, a model was developed with every first order interaction term. Again, insignificant terms were removed to form a reduced model and ANOVA was performed to confirm a lack of difference. All assumptions of linear regression were checked for the multivariate models. Regularization techniques were utilized on a model with interaction in order to reduce overfitting. Random Forrest was then applied in order to allow the computer to learn nonlinear trends in the data. This algorithm underwent grid search to optimize its hyperparameters.

Parameter	Search Space
n_estimators	10,25,30,50,100,200
max_depth	2,3,5,10,20
min_samples_leaf	5,10,20,50,100,200

FIGURE 1: Search space for Random Forest Optimization

Neural networks were tested to determine the efficacy of a computationally expensive model.

```
Model: "sequential_32"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense_167 (Dense)           (None, 64)                448

 dense_168 (Dense)           (None, 64)                4160

 dense_169 (Dense)           (None, 128)               8320

 dense_170 (Dense)           (None, 64)                8256

 dense_171 (Dense)           (None, 32)                2080

 dense_172 (Dense)           (None, 1)                 33

=================================================================
Total params: 23,297
Trainable params: 23,297
Non-trainable params: 0
_________________________________________________________________
```

FIGURE 2: Neural Network Model Summary

Finally, the k-nearest neighbor was optimized using grid search and used to test the efficacy of distance-based modeling.

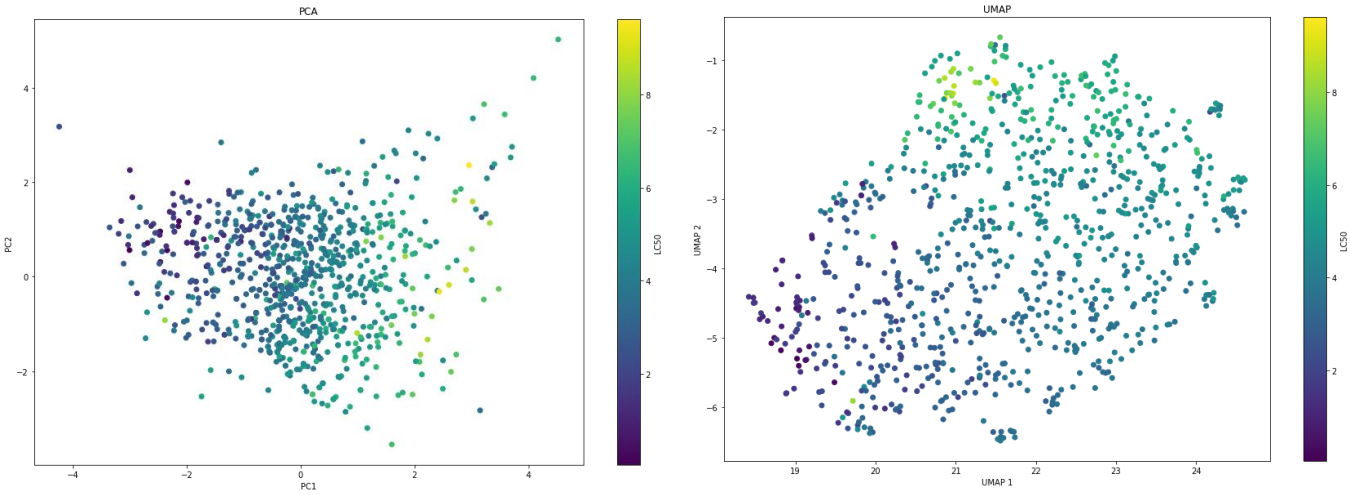
Parameter	Search Space
n_neighbors (k)	1:6
weights	Uniform, Distance
p	1 (Manhattan),2 (Euclidean)

FIGURE 3: Search space for KNN Optimization

All these models were tested against the validation set and Mean Squared Error (MSE), Mean Average Error (MAE), and R^2 scores were reported. Full code and hyperparameters are found in the appendix.

RESULTS

Dimensionality Reduction



FIGURES 4&5: PCA and UMAP on Training Features

Linear Regression

Univariate models were first tested to establish a performance baseline for the multivariate model and determine which variables might be of more importance in the multivariate model. The strongest univariate model was based on MLOGP and had a training $r^2 = 0.423$ and a testing $r^2 = 0.412$. The multivariate model without interaction had higher performance, with a training $r^2 = 0.564$ and testing $r^2 = 0.610$. Partial F-test revealed the

NdssC variable was not significant at $\alpha = 0.05$ and thus a reduced model was tested removing the insignificant variable. ANOVA was performed between the full and reduced model, and there was no indication of statistically significant difference between the full and reduced model. Training $r^2 = 0.563$ and testing $r^2 = 0.607$ for the reduced model. The model with all possible first order interactions was then fitted and produced a training $r^2 = 0.614$ and testing $r^2 = 0.607$. All insignificant terms were removed from the model and ANOVA confirmed there was no statistically significant differences between the full and reduced model at $\alpha = 0.05$. The reduced first order model produced a training $r^2 = 0.603$ and a testing $r^2 = 0.589$. All assumptions of linear regression were validated for the model utilizing diagnostic plots and VIF analysis. All the model summaries, ANOVA, diagnostics, and validation scores are found in the appendix.

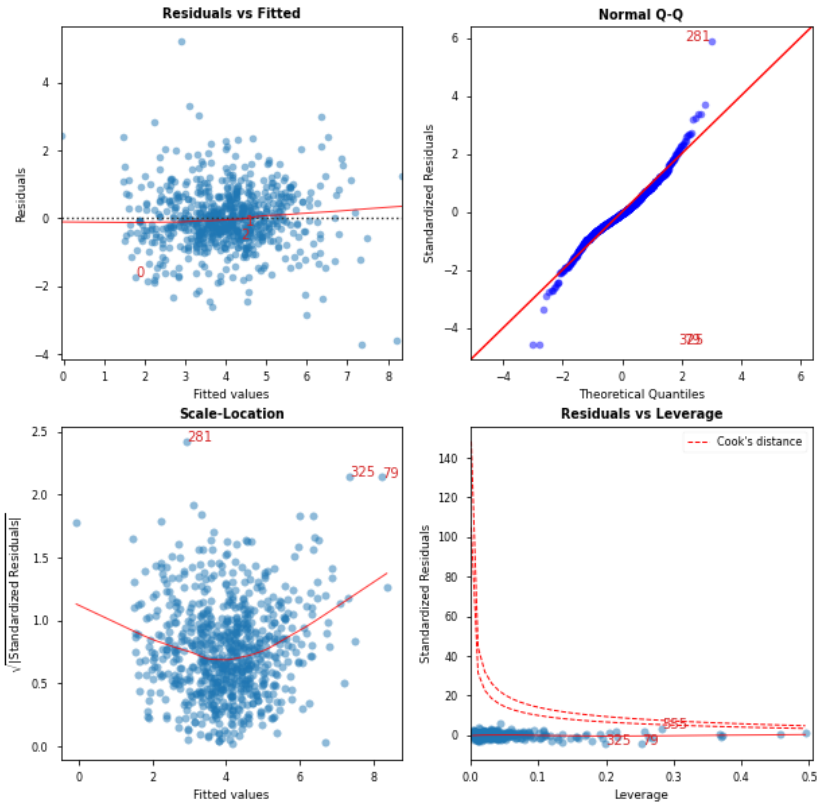


FIGURE 6: Diagnostic Plots for the Full Interaction Model

Regularization

Regularization techniques were applied on the second order regression model with all terms. While adding more features gives the model more information to train, it is prone to

overfitting leading to poor validation results. The goal of regularization is to constrain the model and prevent overfitting while providing more information than simple multivariate regression. Three regularization techniques were utilized: Ridge, Lasso, and Elasticnet. Ridge regularization produced validation $r^2 = 0.620$, Lasso regularization produced validation $r^2 = 0.608$, and Elasticnet produced validation $r^2 = 0.620$. All validation scores are in the appendix.

Random Forest

Random Forest regression was utilized to examine the inclusion of nonlinear trends in the model. Grid search was first utilized to optimize the hyperparameters (FIGURE 1), and resulted in a Random Forest model with n_estimators = 50, max_depth = 20, and min_samples_leaf = 5. The model with the best hyperparameters was then fitted on the training data and validated on the testing set, resulting in a validation $r^2 = 0.653$. All validation scores are in the appendix.

Sequential Neural Network

Sequential neural network was utilized to provide an example of the most computationally intensive model. The neural network used in this work is defined in FIGURE 2, was trained for 100 epochs, and optimized using the rmsprop algorithm. The validation $r^2 = 0.662$. All validation scores are in the appendix.

K-Nearest Neighbor (KNN)

KNN, while not interpretable as a model provides distance based, nonlinear learning. KNN was first optimized using grid search on the search space defined in FIGURE 3. Resulting hyperparameters were k = 5, distance-based weights, and p = 1. Validation $r^2 = 0.711$. All validation scores are in the appendix.

DISCUSSION

Dimensionality Reduction

Dimensionality reduction was employed to better visualize the relations between all 6 features and the target variable. PCA exhibited strong clustering of points with lower LC50 values while points with higher LC50 values were more spread out. Supervised UMAP exhibited the same consolidation of points with low LC50 values but also exhibited a strong cluster of points with high LC50. Since PCA is a fully linear transformation, and UMAP is a nonlinear manifold transformation, it can be deduced that there are certain trends in the data not fully capturable by a linear model. While the original scope of the work was to develop a strong, multivariate linear regression model to predict chemical toxicity, it was decided that further models needed testing in order to develop a strong model for predicting toxicity.

Linear Regression

As suggested by the previous section, there were nonlinear trends in the data that the linear model would be unable to learn. Within the scope of linear regression, two main models were tested, one with interaction, and the other without considering interaction. The models performed standardly, with the model with interaction having higher training scores than the model without interaction. This training success did not carry over to validation, indicating the model with interaction was overfitting the data, as the model without interaction validated much better than the model with interaction. The model with interaction potentially included too extraneous information, such that the model fit the training dataset too well and did not learn the general trend as well as the model provided with less information. Based on validation r^2 and considering potential overfitting, the best linear model is: $E(LC50) = 4.038 + 0.307 * CIC0 + 0.541 * SM1_{Dz} - 0.273 * GATS1i + 0.243 * NdsCH + 0.5562 * MLOGP$.

Regularization

As previously mentioned, the model with interactions was prone to overfitting. To solve this issue, regularization was applied to constrain the coefficients of the model. Both Lasso and Elasticnet were able to increase performance against the validation set compared to not only the overfit interaction model, but the best linear model as well. Regularization allowed for the

consideration of interactions and higher order terms while weighing them less than individual terms, increasing the overall performance of the model without overfitting.

Random Forest

Random Forest, unlike the previous models is not limited by being linear and is able to learn nonlinear trends in the data. The increased validation score combined with the analysis of PCA and UMAP validates the existence of nonlinear trends in the data, indicating the best performing model must be able to learn these trends. While the points in the UMAP are well clustered, Random Forest does not significantly weigh distance, and acts more like a probabilistic model instead, and thus isn't able to learn from the clustering.

Sequential Neural Network

The neural network indicated the added complexity of the nonlinear model could increase performance. As in the case of Random Forest, dense layers of the network do not consider distances. While this model performed the best out of all the models which did not consider distance, its training takes much longer than the random forest and other models. The scale of the problem is also inappropriate for neural networks, as there are only a limited number of chemicals. Neural networks function better when there are millions of points to learn from.

K-Nearest Neighbor

KNN regression, while being computationally less intensive than Random Forest and neural network, factors in distance when learning the data. Because of the clustered nature of the data as evidenced by UMAP, this model is highly appropriate. The validation score of the model suggests this claim is true, as KNN is the best performing model compared to the rest. As KNN lacks interpretability compared to linear regression, there are no coefficients to report. Thus, the model will be reported in terms of the hyperparameters. The best overall model in this study is a 5-Nearest Neighbors model with Manhattan distance and weighing points by distance.

SUGGESTIONS FOR FUTURE RESEARCH

The current scope of the research presents a regression problem, where 6 features are used to predict the value of the target feature—LC50. With future regulations potentially specifying a hard cutoff value for toxicity, there would be more value in classifying a chemicals

safety rather than its specific LC50 value. This would turn the problem into one of binary classification, allowing the use of logistic regression, and increasing the accuracy scores of the model. Points near the decision boundary in logistic regression would weigh the most, thus even if a point regressed poorly, if it were far from the decision boundary, it would matter much less. The model would also return confidence in each point, meaning that only chemicals that pass the model at a set α level would be allowed to be imported. Overall, these findings serve to provide a baseline model for future QSAR toxicity analysis and suggest improvements to future studies—including the use of classification techniques—in order to prevent damage to human health and to the health of the environment caused by toxic chemicals.

REFERENCES

[1] *Regulation (EC) No 1907/2006*. 2006, pp. 1–849

[2] M. Cassotti, D. Ballabio, R. Todeschini, V. Consonni. *A similarity-based QSAR model for predicting acute toxicity towards the fathead minnow (Pimephales promelas)*, SAR and QSAR in Environmental Research (2015) vol. 26, no. 3, 217-243,

[3] C.L. Russom, S.P. Bradbury, S.J. Broderius, D.E. Hammermeister, and R.A. Drummond, *Predicting modes of toxic action from chemical structure: Acute toxicity in the fathead minnow (Pimephales promelas)*, Environ. Toxicol. Chem. 16 (1997), pp. 948–967

[4] L. Michielan, L. Pireddu, M. Floris, and S. Moro, *Support vector machine (SVM) as alternative tool to assign acute aquatic toxicity warning labels to chemicals*, Mol. Inform. 29 (2010), pp. 51–64.

[5] M. Nendza, M. Müller, and A. Wenzel, *Discriminating toxicant classes by mode of action: 4. Baseline and excess toxicity*, SAR QSAR Environ. Res. 25 (2014), pp. 393–405.

[6] A. Levet, C. Bordes, Y. Clément, P. Mignon, H. Chermette, P. Marote, C. Cren-Olivé, and P. Lantéri, *Quantitative structure–activity relationship to predict acute fish toxicity of organic solvents*, Chemosphere 93 (2013), pp. 1094–1103.

[7] W.D. Marzio and M.E. Saenz, Quantitative structure–activity relationship for aromatic hydrocarbons on freshwater fish, Ecotoxicol. Environ. Saf. 59 (2004), pp. 256–262.

[8] K. Rose and L.H. Hall, *E-state modeling of fish toxicity independent of 3D structure information*, SAR QSAR Environ. Res. 14 (2003), pp. 113–129.

[9] T.I. Netzeva, A.O. Aptula, E. Benfenati, M.T.D. Cronin, G. Gini, I. Lessigiarska, U. Maran, M. Vračko, and G. Schüürmann, *Description of the electronic structure of organic chemicals using semiempirical and ab initio methods for development of toxicological QSARs*, J. Chem. Inf. Model. 45 (2005), pp. 106–114.

[10] M. Pavan, T.I. Netzeva, and A.P. Worth, *Validation of a QSAR model for acute toxicity*, SAR QSAR Environ. Res. 17 (2006), pp. 147–171.

[11] K. Roy and R.N. Das, *QSTR with extended topochemical atom (ETA) indices. 15. Development of predictive models for toxicity of organic chemicals against fathead minnow using second-generation ETA indices*, SAR QSAR Environ. Res. 23 (2012), pp. 125–140.

[12] Y.-Y. In, S.-K. Lee, P.-J. Kim, and K.-T. No, *Prediction of acute toxicity to fathead minnow by local model based QSAR and global QSAR approaches*, Bull. Korean Chem. Soc. 33 (2012), pp. 613–619. [10] T. Martin, P. Harten, R. Venkatapathy, and D. Young, T.E.S.T. (Toxicity Estimation Software Tool). U.S. E.P.A., 2012

[13] M. Casalegno, E. Benfenati, and G. Sello, *An automated group contribution method in predicting aquatic toxicity: The diatomic fragment approach*, Chem. Res. Toxicol. 18 (2005), pp. 740–746.

[14] D.V. Eldred, C.L. Weikel, P.C. Jurs, and K.L.E. Kaiser, *Prediction of fathead minnow acute toxicity of organic compounds from molecular structure*, Chem. Res. Toxicol. 12 (1999), pp. 670–678.

[15] M. Hewitt, M.T.D. Cronin, J.C. Madden, P.H. Rowe, C. Johnson, A. Obi, and S.J. Enoch, *Consensus QSAR Models: Do the benefits outweigh the complexity?*, J. Chem. Inf. Model. 47 (2007), pp. 1460–1468.

[16] S. Lozano, M.-P. Halm-Lemeille, A. Lepailleur, S. Rault, and R. Bureau, *Consensus QSAR related to global or MOA models: Application to acute toxicity for fish*, Mol. Inform. 29 (2010), pp. 803–813.

[17] U. Maran, S. Sild, P. Mazzatorta, M. Casalegno, E. Benfenati, and M. Romberg, *Grid computing for the estimation of toxicity: Acute toxicity on fathead minnow (Pimephales promelas), in Distributed, High-Performance and Grid Computing in Computational Biology*, W. Dubitzky, A. Schuster, P. Sloot, M. Schroeder, and M. Romberg, eds., Springer, Berlin, 2007, pp. 60–74

[18] M. Nendza and C.L. Russom, *QSAR modelling of the ERL-D fathead minnow acute toxicity database*, Xenobiotica 21 (1991), pp. 147–170.

[19] E. Papa, F. Villa, and P. Gramatica, *Statistically validated QSARs, based on theoretical descriptors, for modeling aquatic toxicity of organic chemicals in Pimephales promelas (fathead*

minnow), J. Chem. Inf. Model. 45 (2005), pp. 1256–1266. SAR and QSAR in Environmental Research 241

[20] A.P. Toropova, A.A. Toropov, A. Lombardo, A. Roncaglioni, E. Benfenati, and G. Gini, *Coral: QSAR models for acute toxicity in fathead minnow (Pimephales promelas)*, J. Comput. Chem. pp. 1218–1223.

[21] Y. Wang, M. Zheng, J. Xiao, Y. Lu, F. Wang, J. Lu, X. Luo, W. Zhu, H. Jiang, and K. Chen, *Using support vector regression coupled with the genetic algorithm for predicting acute toxicity to the fathead minnow*, SAR QSAR Environ. Res. 21 (2010), pp. 559–570.

[22] P. Mazzatorta, E. Benfenati, C.-D. Neagu, and G. Gini, *Tuning neural and fuzzy-neural networks for toxicity modeling*, J. Chem. Inf. Model. 43 (2003), pp. 513–518.

[23] P. Mazzatorta, M. Vracko, A. Jezierska, and E. Benfenati, *Modeling toxicity by using supervised kohonen neural networks*, J. Chem. Inf. Model. 43 (2003), pp. 485–492.

[24] J. Devillers, *A new strategy for using supervised artificial neural networks in QSAR*, SAR QSAR Environ. Res. 16 (2005), pp. 433–442.

[25] S.P. Niculescu, A. Atkinson, G. Hammond, and M. Lewis, *Using fragment chemistry data mining and probabilistic neural networks in screening chemicals for acute toxicity to the fathead minnow*, SAR QSAR Environ. Res. 15 (2004), pp. 293–309.

[26] McKinney, W. & others, 2010. *Data structures for statistical computing in python*. In Proceedings of the 9th Python in Science Conference. pp. 51–56.

[27] Harris, C.R., Millman, K.J., van der Walt, S.J. et al. *Array programming with NumPy*. Nature 585, 357–362 (2020).

[28] *Scikit-learn: Machine Learning in Python*, Pedregosa et al., JMLR 12, pp. 2825-2830, 2011.

[29] Seabold, Skipper, and Josef Perktold. *statsmodels: Econometric and statistical modeling with python*. Proceedings of the 9th Python in Science Conference. 2010.

[30] J. D. Hunter, *Matplotlib: A 2D Graphics Environment*, Computing in Science & Engineering, vol. 9, no. 3, pp. 90-95, 2007.

[31] M. L. Waskom, *seaborn: statistical data visualization*, Journal of Open Source Software, vol. 6, no. 60, p. 3021, 2021

[32] M. Abadi et al., *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. 2015

[33] McInnes, L., Healy, J., Saul, N., & Großberger, L. (2018). *UMAP: Uniform Manifold Approximation and Projection*. Journal of Open Source Software, 3(29), 861

APPENDIX

1 Load and Analyze Data

[1]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import missingno as msno
from scipy import stats
import sklearn
import statsmodels
import statsmodels.api as sm
import statsmodels.formula.api as smf
from sklearn.metrics import mean_squared_error
from sklearn.metrics import mean_absolute_error
from sklearn.metrics import r2_score
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.neighbors import KNeighborsRegressor
from sklearn.decomposition import PCA
import random
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import KFold
from sklearn.model_selection import GridSearchCV
from sklearn.metrics import make_scorer
from sklearn.pipeline import Pipeline
from numpy import mean
from numpy import std
from sklearn.linear_model import SGDRegressor
from sklearn.linear_model import Ridge
from sklearn.preprocessing import PolynomialFeatures
from sklearn.linear_model import Lasso
from sklearn.linear_model import ElasticNet
from sklearn.ensemble import RandomForestRegressor
import umap
import tensorflow as tf
from tensorflow.keras import Model
from tensorflow.keras import Sequential
```

```
from tensorflow.keras.optimizers import Adam
from sklearn.preprocessing import StandardScaler
from tensorflow.keras.layers import Dense, Dropout
from sklearn.model_selection import train_test_split
from tensorflow.keras.losses import MeanSquaredLogarithmicError
from sklearn.metrics import mean_squared_error
from keras.callbacks import ModelCheckpoint
from statsmodels.stats.outliers_influence import variance_inflation_factor
from sklearn.manifold import TSNE
from statsmodels.tools.tools import maybe_unwrap_results
from statsmodels.graphics.gofplots import ProbPlot
from statsmodels.stats.outliers_influence import variance_inflation_factor
from typing import Type
style_talk = 'seaborn-talk'      #refer to plt.style.available
random.seed(101)
np.random.seed(101)
%matplotlib inline
```

[2]:
```
class Linear_Reg_Diagnostic():
    """
    Diagnostic plots to identify potential problems in a linear regression fit.
    Mainly,
        a. non-linearity of data
        b. Correlation of error terms
        c. non-constant variance
        d. outliers
        e. high-leverage points
        f. collinearity

"""


    def __init__(self,
                 results:  Type[statsmodels.regression.linear_model.
↪RegressionResultsWrapper]) -> None:
        """
        For a linear regression model, generates following diagnostic plots:

        a. residual
        b. qq
        c. scale location and
        d. leverage

        and a table

        e. vif

        Args:
```

```
            results (Type[statsmodels.regression.linear_model.
↪RegressionResultsWrapper]):
                must be instance of statsmodels.regression.linear_model object

        Raises:
            TypeError: if instance does not belong to above object

        Example:
        >>> import numpy as np
        >>> import pandas as pd
        >>> import statsmodels.formula.api as smf
        >>> x = np.linspace(-np.pi, np.pi, 100)
        >>> y = 3*x + 8 + np.random.normal(0,1, 100)
        >>> df = pd.DataFrame({'x':x, 'y':y})
        >>> res = smf.ols(formula= "y ~ x", data=df).fit()
        >>> cls = Linear_Reg_Diagnostic(res)
        >>> cls(plot_context="seaborn-paper")

        In case you do not need all plots you can also independently make an␣
↪individual plot/table
        in following ways

        >>> cls = Linear_Reg_Diagnostic(res)
        >>> cls.residual_plot()
        >>> cls.qq_plot()
        >>> cls.scale_location_plot()
        >>> cls.leverage_plot()
        >>> cls.vif_table()
        """

        if isinstance(results, statsmodels.regression.linear_model.
↪RegressionResultsWrapper) is False:
            raise TypeError("result must be instance of statsmodels.regression.
↪linear_model.RegressionResultsWrapper object")

        self.results = maybe_unwrap_results(results)

        self.y_true = self.results.model.endog
        self.y_predict = self.results.fittedvalues
        self.xvar = self.results.model.exog
        self.xvar_names = self.results.model.exog_names

        self.residual = np.array(self.results.resid)
        influence = self.results.get_influence()
        self.residual_norm = influence.resid_studentized_internal
        self.leverage = influence.hat_matrix_diag
        self.cooks_distance = influence.cooks_distance[0]
```

```
        self.nparams = len(self.results.params)

    def __call__(self, plot_context='seaborn-paper'):
        # print(plt.style.available)
        with plt.style.context(plot_context):
            fig, ax = plt.subplots(nrows=2, ncols=2, figsize=(10,10))
            self.residual_plot(ax=ax[0,0])
            self.qq_plot(ax=ax[0,1])
            self.scale_location_plot(ax=ax[1,0])
            self.leverage_plot(ax=ax[1,1])
            plt.show()

        self.vif_table()
        return fig, ax


    def residual_plot(self, ax=None):
        """
        Residual vs Fitted Plot

        Graphical tool to identify non-linearity.
        (Roughly) Horizontal red line is an indicator that the residual has a␣
↪linear pattern
        """
        if ax is None:
            fig, ax = plt.subplots()

        sns.residplot(
            x=self.y_predict,
            y=self.residual,
            lowess=True,
            scatter_kws={'alpha': 0.5},
            line_kws={'color': 'red', 'lw': 1, 'alpha': 0.8},
            ax=ax)

        # annotations
        residual_abs = np.abs(self.residual)
        abs_resid = np.flip(np.sort(residual_abs))
        abs_resid_top_3 = abs_resid[:3]
        for i, _ in enumerate(abs_resid_top_3):
            ax.annotate(
                i,
                xy=(self.y_predict[i], self.residual[i]),
                color='C3')

        ax.set_title('Residuals vs Fitted', fontweight="bold")
        ax.set_xlabel('Fitted values')
```

```
        ax.set_ylabel('Residuals')
        return ax

    def qq_plot(self, ax=None):
        """
        Standarized Residual vs Theoretical Quantile plot

        Used to visually check if residuals are normally distributed.
        Points spread along the diagonal line will suggest so.
        """
        if ax is None:
            fig, ax = plt.subplots()

        QQ = ProbPlot(self.residual_norm)
        QQ.qqplot(line='45', alpha=0.5, lw=1, ax=ax)

        # annotations
        abs_norm_resid = np.flip(np.argsort(np.abs(self.residual_norm)), 0)
        abs_norm_resid_top_3 = abs_norm_resid[:3]
        for r, i in enumerate(abs_norm_resid_top_3):
            ax.annotate(
                i,
                xy=(np.flip(QQ.theoretical_quantiles, 0)[r], self.
→residual_norm[i]),
                ha='right', color='C3')

        ax.set_title('Normal Q-Q', fontweight="bold")
        ax.set_xlabel('Theoretical Quantiles')
        ax.set_ylabel('Standardized Residuals')
        return ax

    def scale_location_plot(self, ax=None):
        """
        Sqrt(Standarized Residual) vs Fitted values plot

        Used to check homoscedasticity of the residuals.
        Horizontal line will suggest so.
        """
        if ax is None:
            fig, ax = plt.subplots()

        residual_norm_abs_sqrt = np.sqrt(np.abs(self.residual_norm))

        ax.scatter(self.y_predict, residual_norm_abs_sqrt, alpha=0.5);
        sns.regplot(
            x=self.y_predict,
            y=residual_norm_abs_sqrt,
```

```
            scatter=False, ci=False,
            lowess=True,
            line_kws={'color': 'red', 'lw': 1, 'alpha': 0.8},
            ax=ax)

        # annotations
        abs_sq_norm_resid = np.flip(np.argsort(residual_norm_abs_sqrt), 0)
        abs_sq_norm_resid_top_3 = abs_sq_norm_resid[:3]
        for i in abs_sq_norm_resid_top_3:
            ax.annotate(
                i,
                xy=(self.y_predict[i], residual_norm_abs_sqrt[i]),
                color='C3')
        ax.set_title('Scale-Location', fontweight="bold")
        ax.set_xlabel('Fitted values')
        ax.set_ylabel(r'$\sqrt{|\mathrm{Standardized\ Residuals}|}$');
        return ax

    def leverage_plot(self, ax=None):
        """
        Residual vs Leverage plot

        Points falling outside Cook's distance curves are considered␣
↪observation that can sway the fit
         aka are influential.
        Good to have none outside the curves.
        """
        if ax is None:
            fig, ax = plt.subplots()

        ax.scatter(
            self.leverage,
            self.residual_norm,
            alpha=0.5);

        sns.regplot(
            x=self.leverage,
            y=self.residual_norm,
            scatter=False,
            ci=False,
            lowess=True,
            line_kws={'color': 'red', 'lw': 1, 'alpha': 0.8},
            ax=ax)

        # annotations
        leverage_top_3 = np.flip(np.argsort(self.cooks_distance), 0)[:3]
        for i in leverage_top_3:
```

```
            ax.annotate(
                i,
                xy=(self.leverage[i], self.residual_norm[i]),
                color = 'C3')

        xtemp, ytemp = self._cooks_dist_line(0.5) # 0.5 line
        ax.plot(xtemp, ytemp, label="Cook's distance", lw=1, ls='--',␣
↪color='red')
        xtemp, ytemp = self._cooks_dist_line(1) # 1 line
        ax.plot(xtemp, ytemp, lw=1, ls='--', color='red')

        ax.set_xlim(0, max(self.leverage)+0.01)
        ax.set_title('Residuals vs Leverage', fontweight="bold")
        ax.set_xlabel('Leverage')
        ax.set_ylabel('Standardized Residuals')
        ax.legend(loc='upper right')
        return ax

    def vif_table(self):
        """
        VIF table

        VIF, the variance inflation factor, is a measure of multicollinearity.
        VIF > 5 for a variable indicates that it is highly collinear with the
        other input variables.
        """
        vif_df = pd.DataFrame()
        vif_df["Features"] = self.xvar_names
        vif_df["VIF Factor"] = [variance_inflation_factor(self.xvar, i) for i␣
↪in range(self.xvar.shape[1])]

        print(vif_df
                .sort_values("VIF Factor")
                .round(2))


    def __cooks_dist_line(self, factor):
        """
        Helper function for plotting Cook's distance curves
        """
        p = self.nparams
        formula = lambda x: np.sqrt((factor * p * (1 - x)) / x)
        x = np.linspace(0.001, max(self.leverage), 50)
        y = formula(x)
        return x, y
```

[3]:
```
df = pd.read_csv("qsar_fish_toxicity.csv", delimiter = ";", names = ["CIC0",
↪"SM1_Dz", "GATS1i", "NdsCH", "NdssC", "MLOGP", "LC50"])
df.shape
```

[3]: (908, 7)

[4]:
```
df.head()
```

[4]:

	CIC0	SM1_Dz	GATS1i	NdsCH	NdssC	MLOGP	LC50
0	3.260	0.829	1.676	0	1	1.453	3.770
1	2.189	0.580	0.863	0	0	1.348	3.115
2	2.125	0.638	0.831	0	0	1.348	3.531
3	3.027	0.331	1.472	1	0	1.807	3.510
4	2.094	0.827	0.860	0	0	1.886	5.390

[5]:
```
df.tail()
```

[5]:

	CIC0	SM1_Dz	GATS1i	NdsCH	NdssC	MLOGP	LC50
903	2.801	0.728	2.226	0	2	0.736	3.109
904	3.652	0.872	0.867	2	3	3.983	4.040
905	3.763	0.916	0.878	0	6	2.918	4.818
906	2.831	1.393	1.077	0	1	0.906	5.317
907	4.057	1.032	1.183	1	3	4.754	8.201

[6]:
```
df.describe()
```

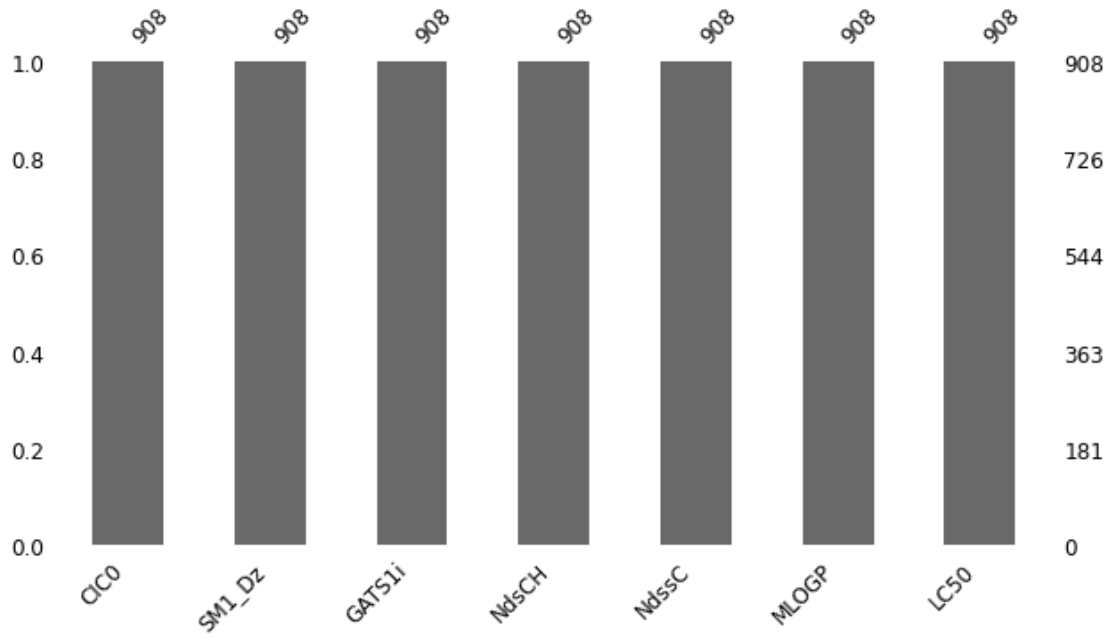
[6]:

	CIC0	SM1_Dz	GATS1i	NdsCH	NdssC	MLOGP	\
count	908.000000	908.000000	908.000000	908.000000	908.000000	908.000000	
mean	2.898129	0.628468	1.293591	0.229075	0.485683	2.109285	
std	0.756088	0.428459	0.394303	0.605335	0.861279	1.433181	
min	0.667000	0.000000	0.396000	0.000000	0.000000	-2.884000	
25%	2.347000	0.223000	0.950750	0.000000	0.000000	1.209000	
50%	2.934000	0.570000	1.240500	0.000000	0.000000	2.127000	
75%	3.407000	0.892750	1.562250	0.000000	1.000000	3.105000	
max	5.926000	2.171000	2.920000	4.000000	6.000000	6.515000	

	LC50
count	908.000000
mean	4.064431
std	1.455698
min	0.053000
25%	3.151750
50%	3.987500
75%	4.907500
max	9.612000

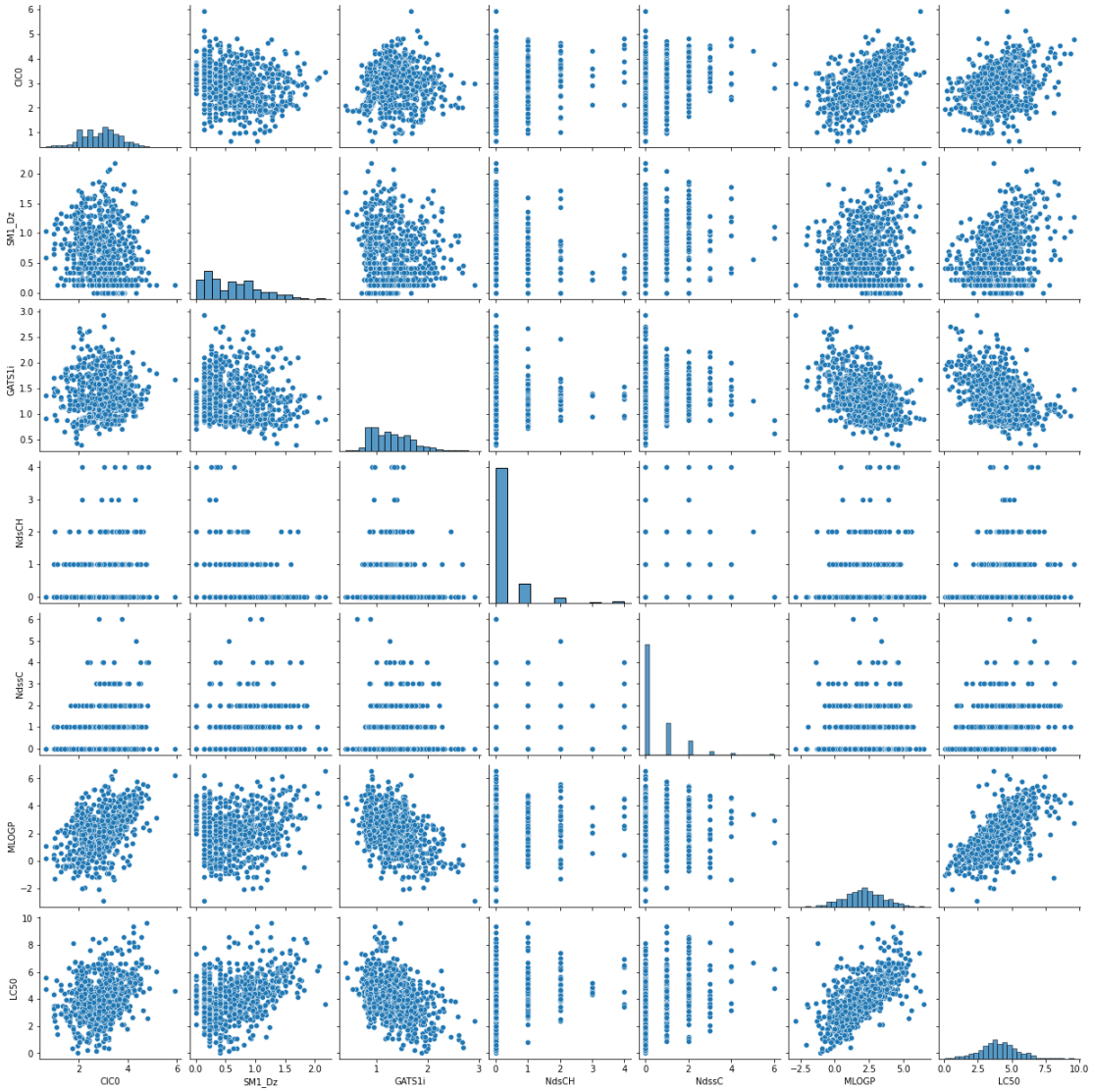
[7]:
```
msno.bar(df, figsize=(10,5), fontsize=12)
```

[7] : <AxesSubplot:>

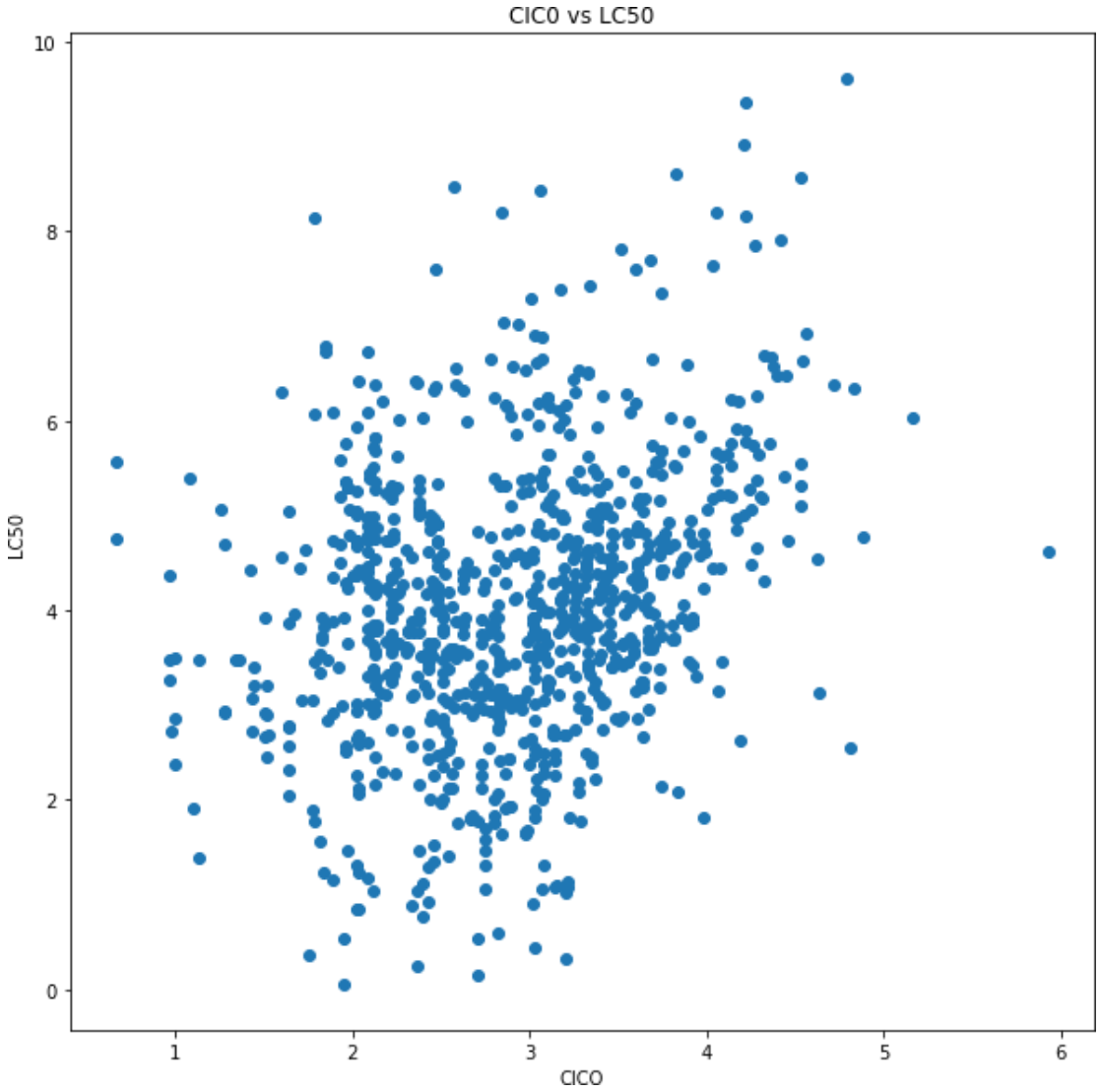


[8]:
```
sns.pairplot(df)
```

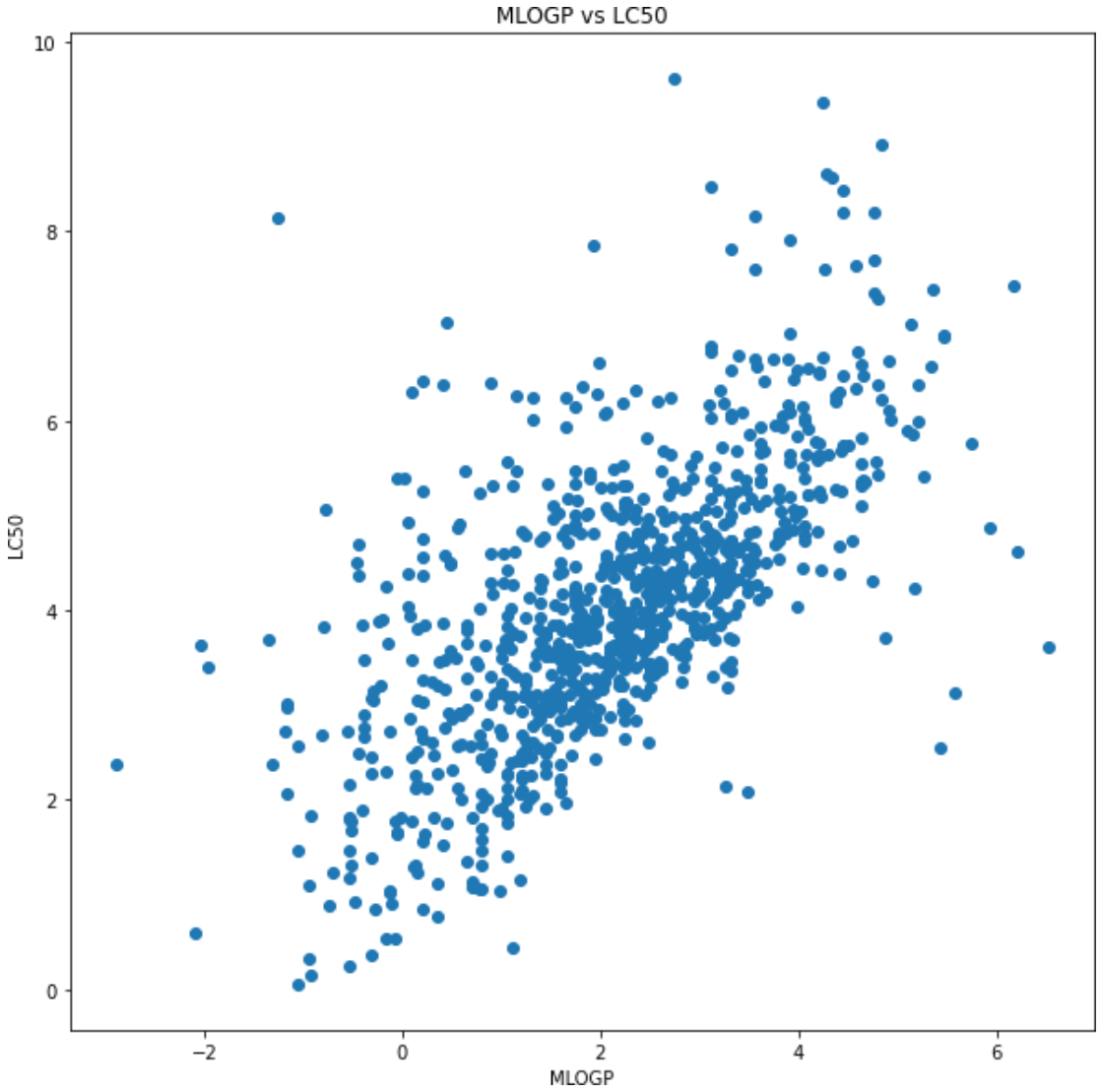
[8] : <seaborn.axisgrid.PairGrid at 0x29ef9575550>



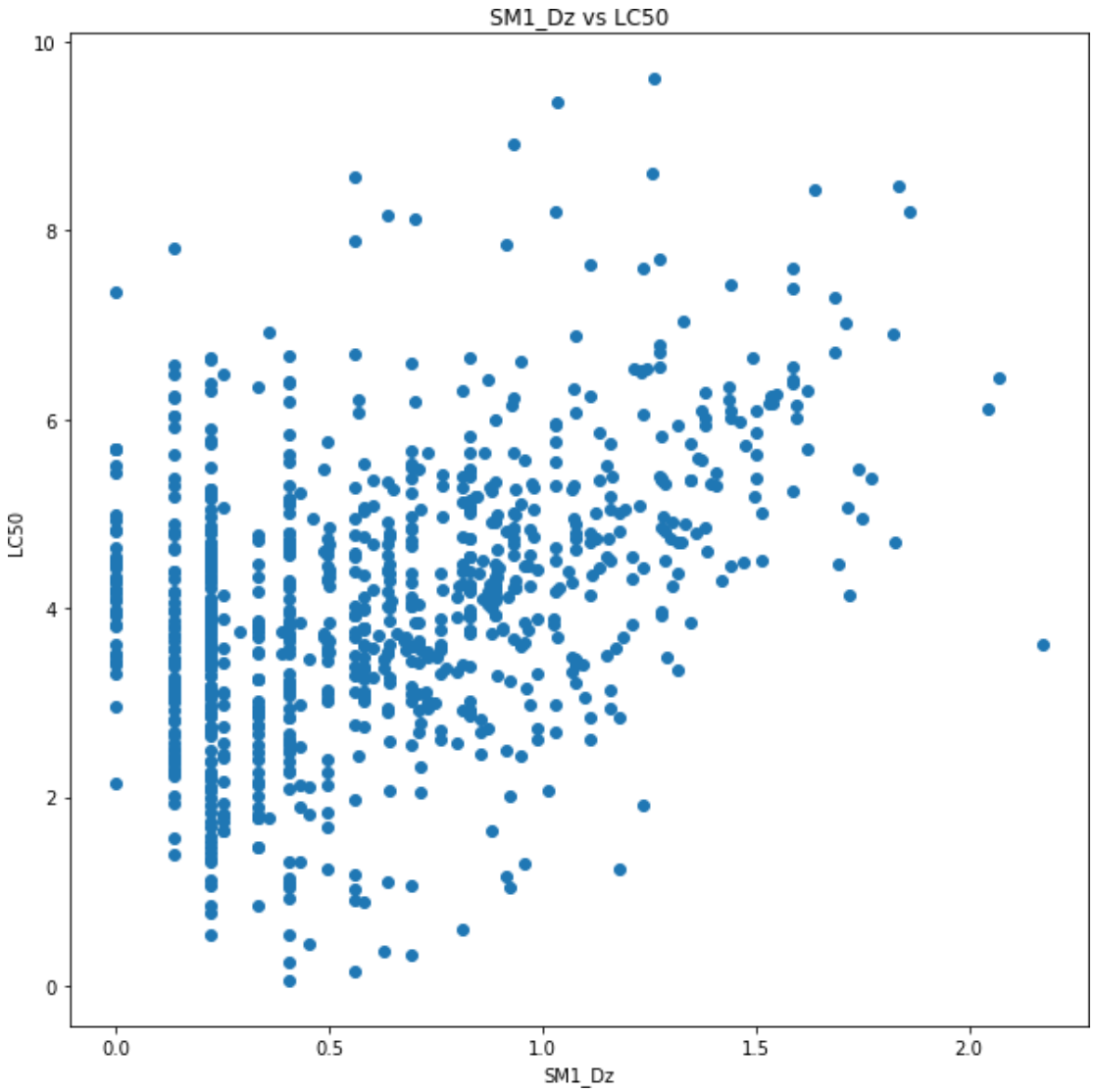
[9]:
```
plt.figure(figsize  =  (10,10))
plot = plt.scatter(df["CIC0"], df["LC50"])
plt.xlabel("CICO")
plt.ylabel("LC50")
plt.title("CIC0 vs LC50")
plt.show()
```



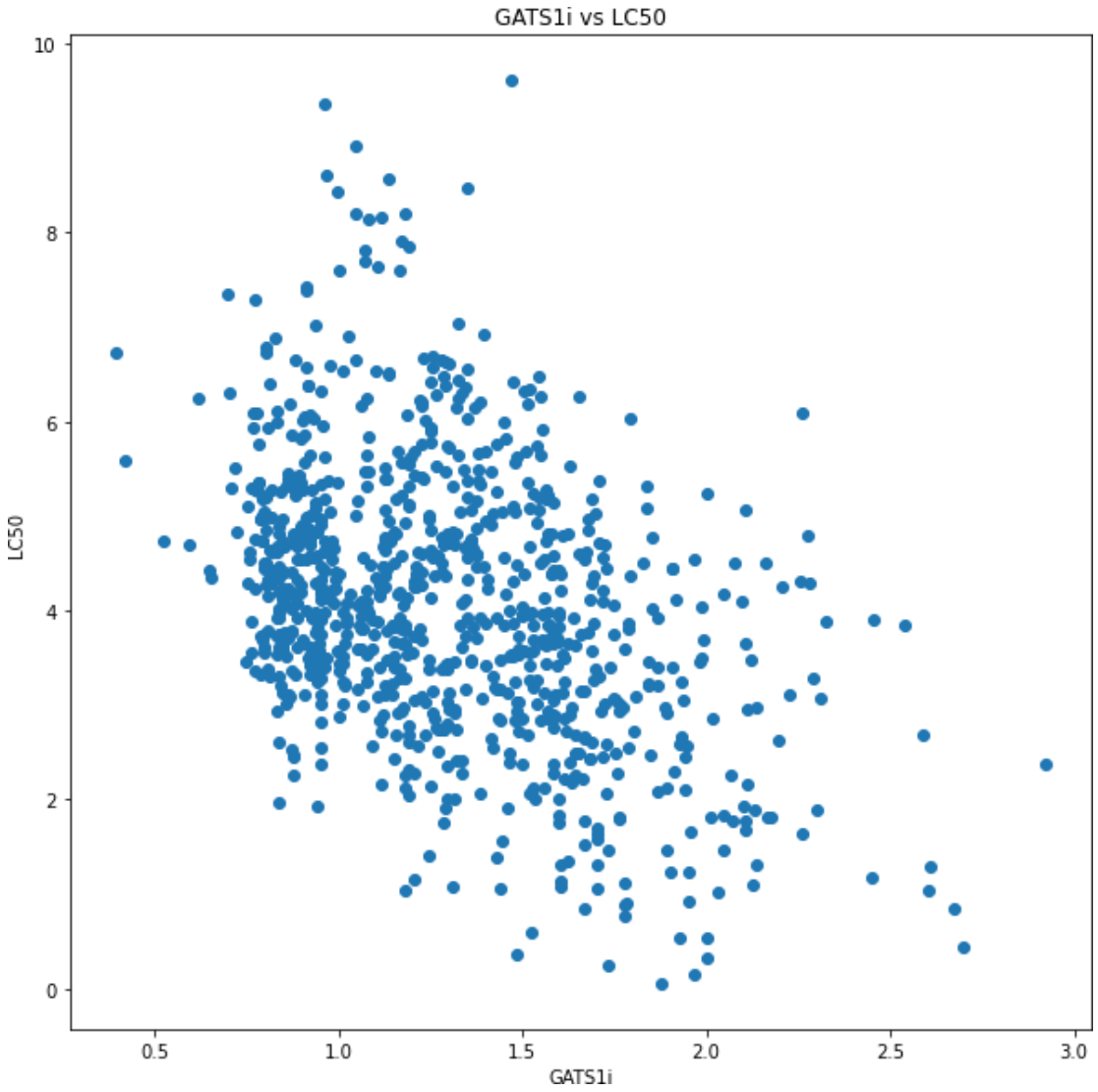
[10]:
```
plt.figure(figsize = (10,10))
plot = plt.scatter(df["MLOGP"], df["LC50"])
plt.xlabel("MLOGP")
plt.ylabel("LC50")
plt.title("MLOGP vs LC50")
plt.show()
```



[11]:
```
plt.figure(figsize = (10,10))
plot = plt.scatter(df["SM1_Dz"], df["LC50"])
plt.xlabel("SM1_Dz")
plt.ylabel("LC50")
plt.title("SM1_Dz vs LC50")
plt.show()
```



[12]:
```
plt.figure(figsize = (10,10))
plot = plt.scatter(df["GATS1i"], df["LC50"])
plt.xlabel("GATS1i")
plt.ylabel("LC50")
plt.title("GATS1i vs LC50")
plt.show()
```



[13]:
```
sns.pairplot(df, hue = "NdsCH")
```

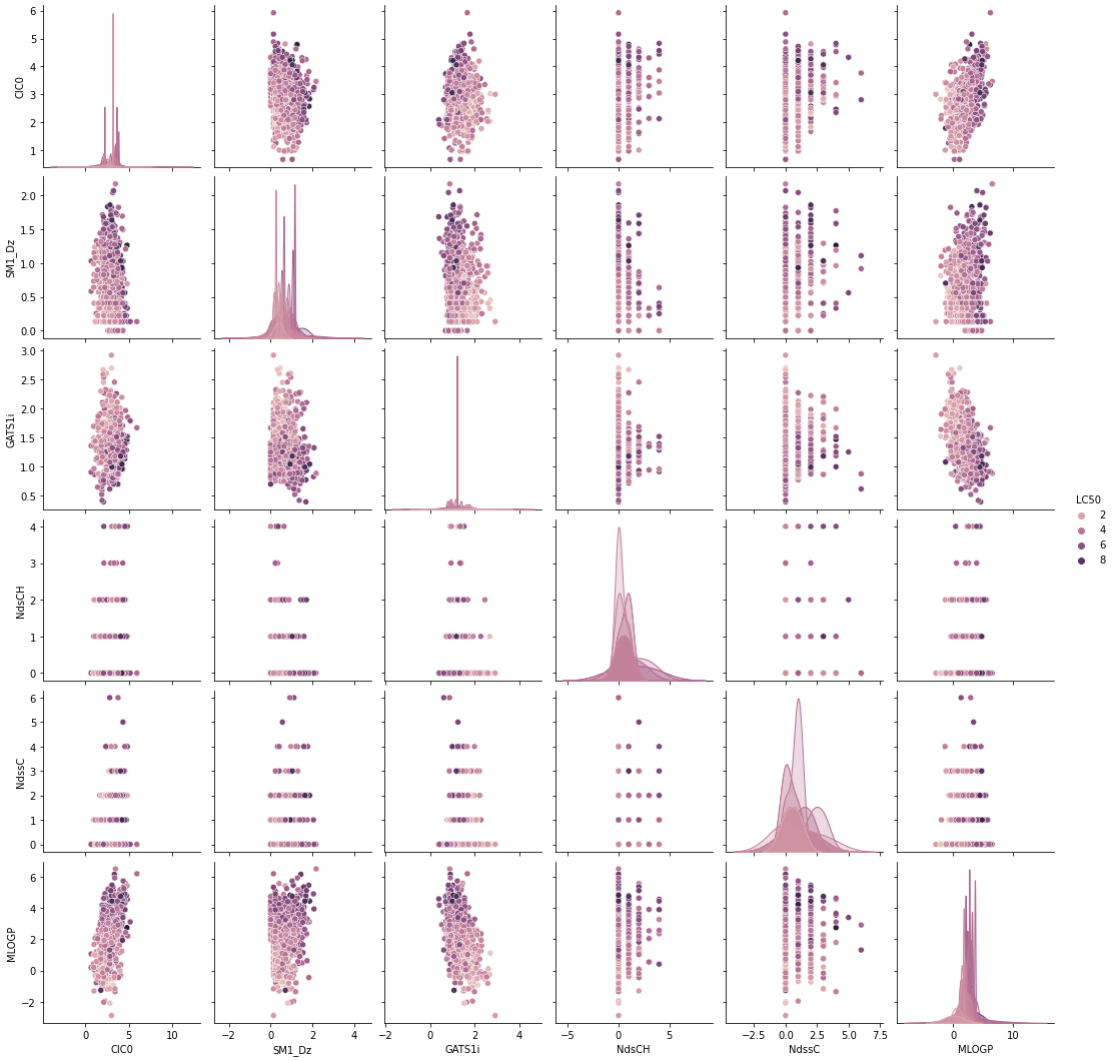
[13]: <seaborn.axisgrid.PairGrid at 0x29efc718b50>

[14]:
```
sns.pairplot(df, hue = "NdssC")
```

[14] : <seaborn.axisgrid.PairGrid at 0x29efcaa7400>

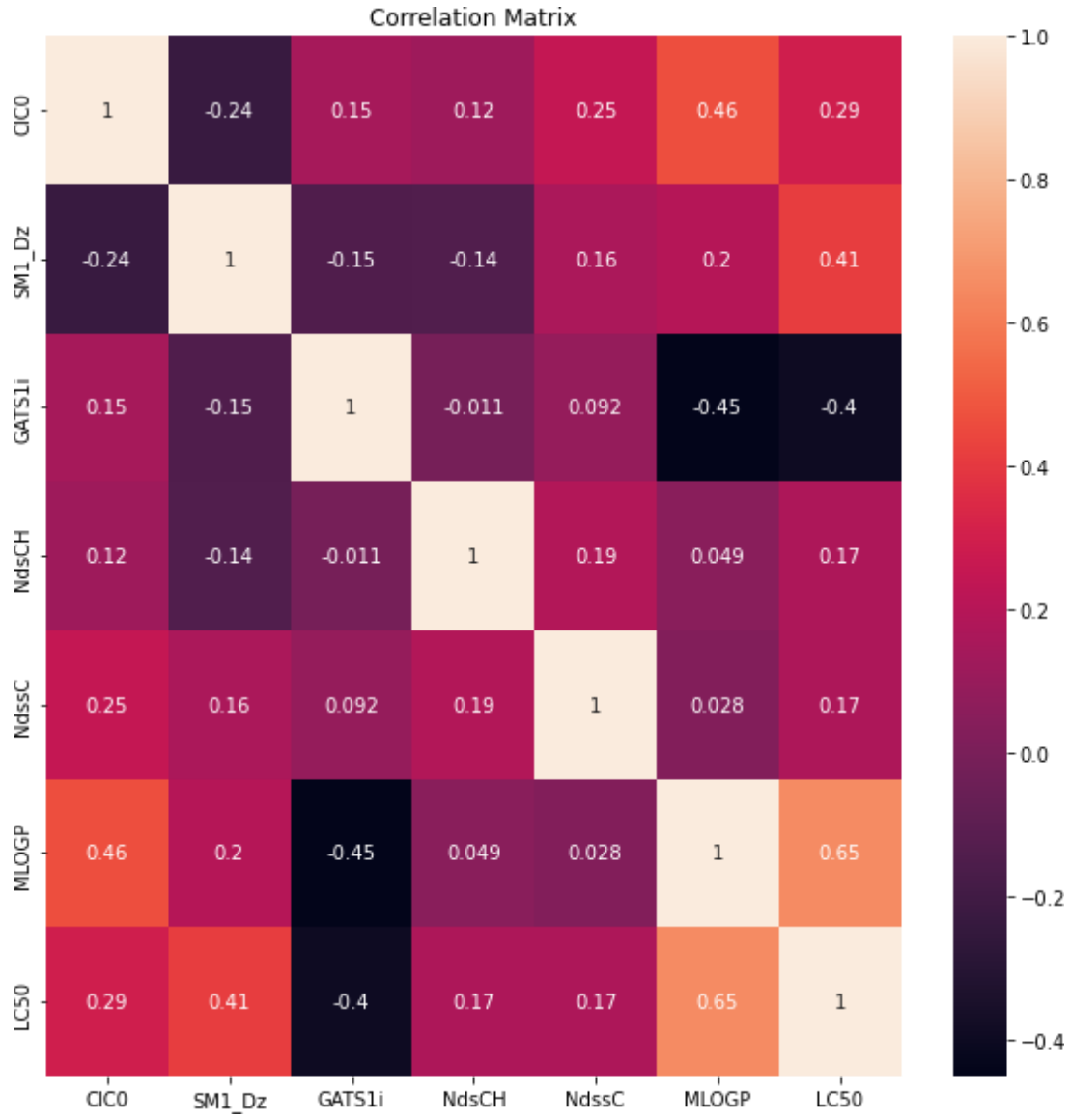
[15]:
```
sns.pairplot(df, hue = "LC50")
```

[15] : <seaborn.axisgrid.PairGrid at 0x29efeb51280>



[16]:
```
corr = df.corr()
plt.figure(figsize = (10,10))
sns.heatmap(corr, annot = True).set(title = "Correlation Matrix")
```

[16] : [Text(0.5, 1.0, 'Correlation Matrix')]

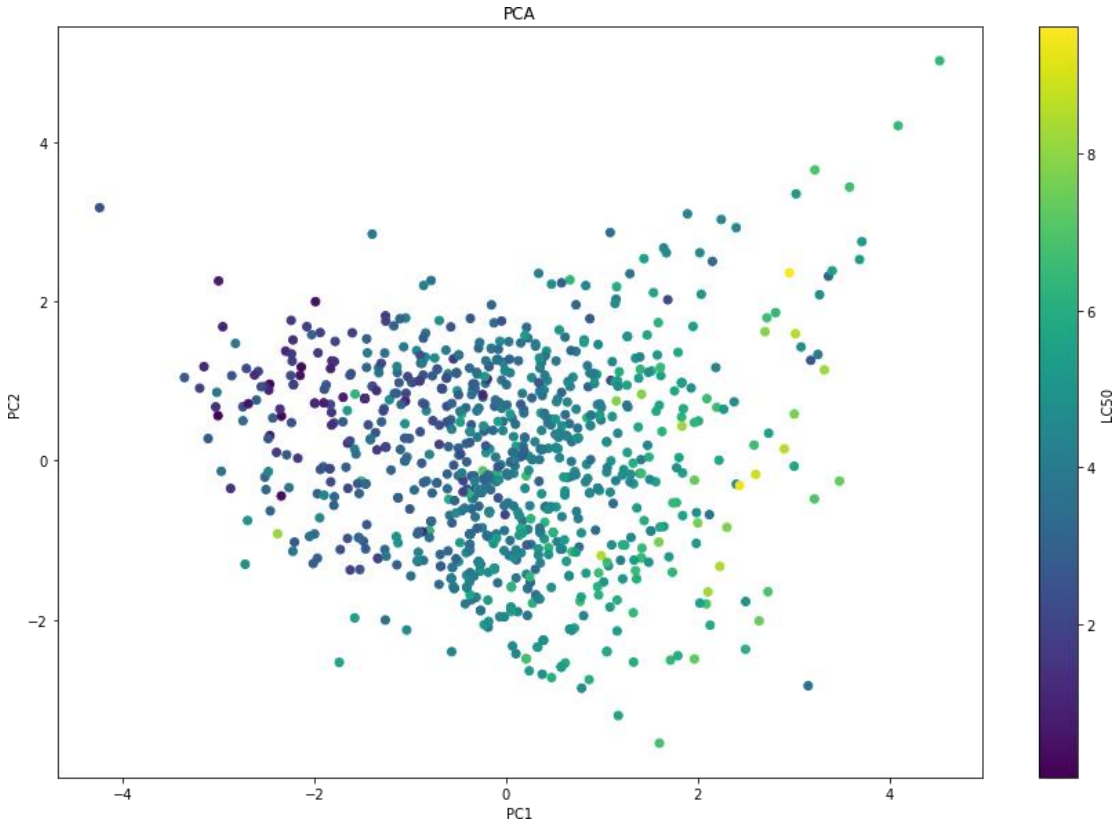


2 Normalize, Split the Data, and Perform Dimensionality Reduction

[17]:
```
X = df.iloc[:, 0:6]
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)
df_scaled = df
df_scaled.iloc[:, 0:6] = X_scaled
Train, Test = train_test_split(df_scaled, test_size=0.2, random_state=101)
X_test = Test.iloc[:, 0:6]
```

```
y_test = Test.iloc[:, 6:]
X_train = Train.iloc[:, 0:6]
y_train = Train.iloc[:, 6:]
```

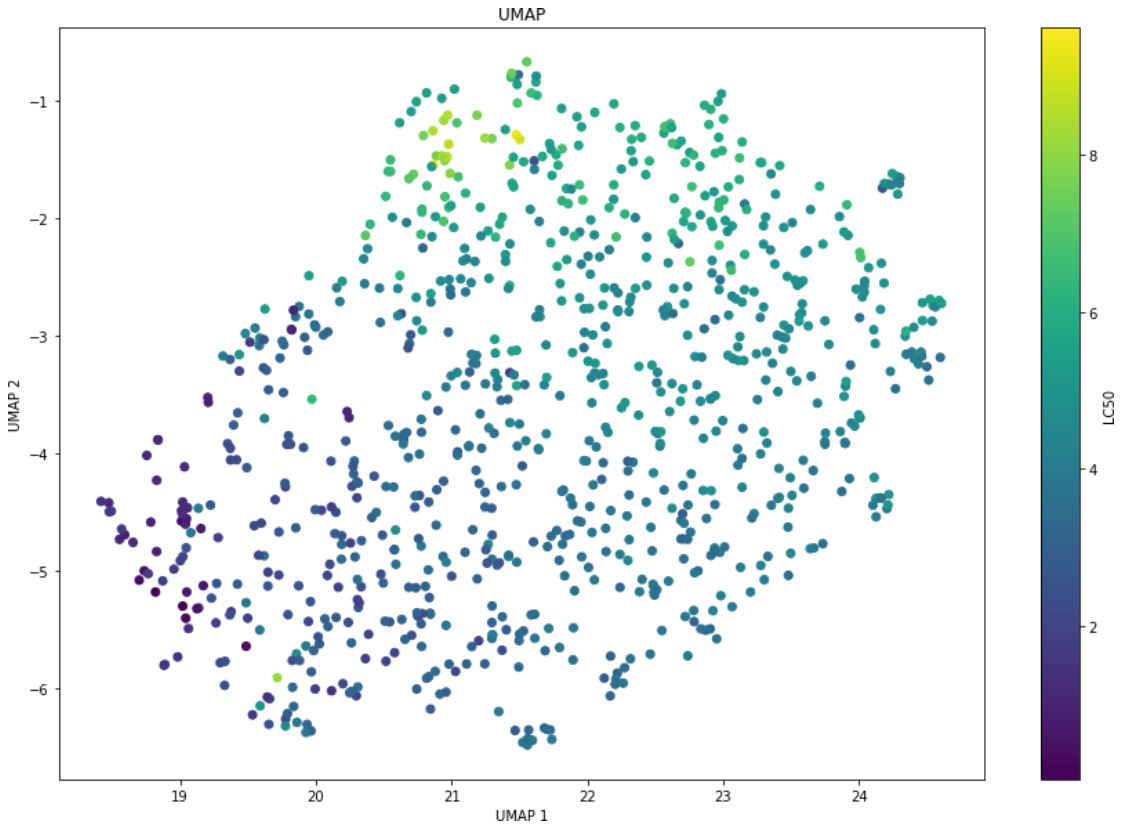
[81]:
```
plt.figure(figsize = (15,10))
pca = PCA(n_components=2)
pc = pca.fit_transform(X_scaled)
plot = plt.scatter(pc[:,0], pc[:,1], c=df.iloc[:, 6:].values)
plt.xlabel("PC1")
plt.ylabel("PC2")
plt.title("PCA")
plt.colorbar(plot, label = "LC50")
plt.show()
```



[74]:
```
reducer = umap.UMAP(target_metric='l1', random_state=101)
embedding = reducer.fit_transform(X_scaled, y = df.iloc[:, 6:].values)
```

[75]:
```
plt.figure(figsize = (15,10))
plot = plt.scatter(embedding[:,0], embedding[:,1], c=df.iloc[:, 6:].values)
plt.title("UMAP")
plt.xlabel("UMAP 1")
```

```
plt.ylabel("UMAP 2")
plt.colorbar(plot, label = "LC50")
plt.show()
```



3 Univariate Model Exploration

3.1 CIC0 vs LC50

[15]:
```
mod = smf.ols(formula = "LC50~CIC0", data = Train).fit()
print(mod.summary())
table = sm.stats.anova_lm(mod, typ=1)
print(table)
y_pred = mod.predict(X_test)
mae = mean_absolute_error(y_test, y_pred)
mse = mean_squared_error(y_test, y_pred)
rmse = mean_squared_error(y_test, y_pred, squared = False)
r2 = r2_score(y_test, y_pred)
print("MAE: {}\nMSE: {}\nRMSE: {}\nR_squared: {}".format(mae, mse, rmse, r2))
```

```
                            OLS Regression Results
==============================================================================
```

Dep. Variable:	LC50	R-squared:	0.099
Model:	OLS	Adj. R-squared:	0.098
Method:	Least Squares	F-statistic:	79.97
Date:	Wed, 27 Jul 2022	Prob (F-statistic):	3.14e-18
Time:	13:00:21	Log-Likelihood:	-1257.0
No. Observations:	726	AIC:	2518.
Df Residuals:	724	BIC:	2527.
Df Model:	1		
Covariance Type:	nonrobust		

	coef	std err	t	P>|t|	[0.025	0.975]
Intercept	4.0056	0.051	78.850	0.000	3.906	4.105
CIC0	0.4496	0.050	8.943	0.000	0.351	0.548

Omnibus:	16.273	Durbin-Watson:	1.946
Prob(Omnibus):	0.000	Jarque-Bera (JB):	19.812
Skew:	0.269	Prob(JB):	4.99e-05
Kurtosis:	3.605	Cond. No.	1.01

Notes:
[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.

	df	sum_sq	mean_sq	F	PR(>F)
CIC0	1.0	149.824163	149.824163	79.968749	3.137657e-18
Residual	724.0	1356.438551	1.873534	NaN	NaN

MAE: 1.1518681199451974
MSE: 2.228243869070734
RMSE: 1.4927303403732148
R_squared: -0.0032098309987493856

3.2 SM1_Dz vs LC50

[16]:
```
mod = smf.ols(formula = "LC50~SM1_Dz", data = Train).fit()
print(mod.summary())
table = sm.stats.anova_lm(mod, typ=1)
print(table)
y_pred = mod.predict(X_test)
mae = mean_absolute_error(y_test, y_pred)
mse = mean_squared_error(y_test, y_pred)
rmse = mean_squared_error(y_test, y_pred, squared = False)
r2 = r2_score(y_test, y_pred)
print("MAE: {}\nMSE: {}\nRMSE: {}\nR_squared: {}".format(mae, mse, rmse, r2))
```

OLS Regression Results

Dep. Variable:	LC50	R-squared:	0.151

Model:	OLS	Adj. R-squared:	0.150
Method:	Least Squares	F-statistic:	129.0
Date:	Wed, 27 Jul 2022	Prob (F-statistic):	1.25e-27
Time:	13:00:21	Log-Likelihood:	-1235.6
No. Observations:	726	AIC:	2475.
Df Residuals:	724	BIC:	2484.
Df Model:	1		
Covariance Type:	nonrobust		

	coef	std err	t	P>|t|	[0.025	0.975]
Intercept	4.0165	0.049	81.432	0.000	3.920	4.113
SM1_Dz	0.5762	0.051	11.359	0.000	0.477	0.676

Omnibus:	12.737	Durbin-Watson:	1.987
Prob(Omnibus):	0.002	Jarque-Bera (JB):	17.750
Skew:	0.169	Prob(JB):	0.000140
Kurtosis:	3.687	Cond. No.	1.03

Notes:
[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.

	df	sum_sq	mean_sq	F	PR(>F)
SM1_Dz	1.0	227.822418	227.822418	129.019268	1.250371e-27
Residual	724.0	1278.440295	1.765802	NaN	NaN

MAE: 0.9679574343670176
MSE: 1.7668341949514315
RMSE: 1.3292231546852589
R_squared: 0.2045280775935706

3.3 GATS1i vs LC50

[17]:
```
mod = smf.ols(formula = "LC50~GATS1i", data = Train).fit()
print(mod.summary())
table = sm.stats.anova_lm(mod, typ=1)
print(table)
y_pred = mod.predict(X_test)
mae = mean_absolute_error(y_test, y_pred)
mse = mean_squared_error(y_test, y_pred)
rmse = mean_squared_error(y_test, y_pred, squared = False)
r2 = r2_score(y_test, y_pred)
print("MAE: {}\nMSE: {}\nRMSE: {}\nR_squared: {}".format(mae, mse, rmse, r2))
```

OLS Regression Results

Dep. Variable:	LC50	R-squared:	0.141
Model:	OLS	Adj. R-squared:	0.140

Method:	Least Squares	F-statistic:	118.7
Date:	Wed, 27 Jul 2022	Prob (F-statistic):	1.07e-25
Time:	13:00:21	Log-Likelihood:	-1240.0
No. Observations:	726	AIC:	2484.
Df Residuals:	724	BIC:	2493.
Df Model:	1		
Covariance Type:	nonrobust		

	coef	std err	t	P>|t|	[0.025	0.975]
Intercept	4.0170	0.050	80.946	0.000	3.920	4.114
GATS1i	-0.5417	0.050	-10.894	0.000	-0.639	-0.444

Omnibus:	38.807	Durbin-Watson:	1.998
Prob(Omnibus):	0.000	Jarque-Bera (JB):	47.144
Skew:	0.514	Prob(JB):	5.79e-11
Kurtosis:	3.708	Cond. No.	1.02

Notes:
[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.

	df	sum_sq	mean_sq	F	PR(>F)
GATS1i	1.0	212.150803	212.150803	118.689257	1.065895e-25
Residual	724.0	1294.111910	1.787447	NaN	NaN

MAE: 1.0042377587990872
MSE: 1.7954857477079382
RMSE: 1.339957367869567
R_squared: 0.1916284485189962

3.4 NdsCH vs LC50

[18]:
```
mod = smf.ols(formula = "LC50~NdsCH", data = Train).fit()
print(mod.summary())
table = sm.stats.anova_lm(mod, typ=1)
print(table)
y_pred = mod.predict(X_test)
mae = mean_absolute_error(y_test, y_pred)
mse = mean_squared_error(y_test, y_pred)
rmse = mean_squared_error(y_test, y_pred, squared = False)
r2 = r2_score(y_test, y_pred)
print("MAE: {}\nMSE: {}\nRMSE: {}\nR_squared: {}".format(mae, mse, rmse, r2))
```

OLS Regression Results

Dep. Variable:	LC50	R-squared:	0.035
Model:	OLS	Adj. R-squared:	0.034
Method:	Least Squares	F-statistic:	26.32

Date:	Wed, 27 Jul 2022	Prob (F-statistic):	3.73e-07
Time:	13:00:21	Log-Likelihood:	-1282.1
No. Observations:	726	AIC:	2568.
Df Residuals:	724	BIC:	2577.
Df Model:	1		
Covariance Type:	nonrobust		

	coef	std err	t	P>|t|	[0.025	0.975]
Intercept	4.0059	0.053	76.178	0.000	3.903	4.109
NdsCH	0.2618	0.051	5.130	0.000	0.162	0.362

Omnibus:	20.415	Durbin-Watson:	1.960
Prob(Omnibus):	0.000	Jarque-Bera (JB):	26.860
Skew:	0.292	Prob(JB):	1.47e-06
Kurtosis:	3.739	Cond. No.	1.03

Notes:
[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.

	df	sum_sq	mean_sq	F	PR(>F)
NdsCH	1.0	52.829295	52.829295	26.315901	3.726305e-07
Residual	724.0	1453.433418	2.007505	NaN	NaN

MAE: 1.1750910409244002
MSE: 2.2797661983049387
RMSE: 1.5098894655917494
R_squared: -0.026406442429465216

3.5 NdssC vs LC50

[19]:
```
mod = smf.ols(formula = "LC50~NdssC", data = Train).fit()
print(mod.summary())
table = sm.stats.anova_lm(mod, typ=1)
print(table)
y_pred = mod.predict(X_test)
mae = mean_absolute_error(y_test, y_pred)
mse = mean_squared_error(y_test, y_pred)
rmse = mean_squared_error(y_test, y_pred, squared = False)
r2 = r2_score(y_test, y_pred)
print("MAE: {}\nMSE: {}\nRMSE: {}\nR_squared: {}".format(mae, mse, rmse, r2))
```

OLS Regression Results

Dep. Variable:	LC50	R-squared:	0.023
Model:	OLS	Adj. R-squared:	0.022
Method:	Least Squares	F-statistic:	17.03
Date:	Wed, 27 Jul 2022	Prob (F-statistic):	4.10e-05

Time:	13:00:21	Log-Likelihood:	-1286.6
No. Observations:	726	AIC:	2577.
Df Residuals:	724	BIC:	2586.
Df Model:	1		
Covariance Type:	nonrobust		

	coef	std err	t	P>|t|	[0.025	0.975]
Intercept	4.0051	0.053	75.686	0.000	3.901	4.109
NdssC	0.2227	0.054	4.127	0.000	0.117	0.329

Omnibus:	8.540	Durbin-Watson:	1.973
Prob(Omnibus):	0.014	Jarque-Bera (JB):	9.901
Skew:	0.169	Prob(JB):	0.00708
Kurtosis:	3.461	Cond. No.	1.02

Notes:
[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.

	df	sum_sq	mean_sq	F	PR(>F)
NdssC	1.0	34.623039	34.623039	17.033436	0.000041
Residual	724.0	1471.639674	2.032651	NaN	NaN

MAE: 1.149265041501467
MSE: 2.18208473043618
RMSE: 1.477188116130163
R_squared: 0.01757214098887272

3.6 MLOGP vs LC50

[20]:
```
mod = smf.ols(formula = "LC50~MLOGP", data = Train).fit()
print(mod.summary())
table = sm.stats.anova_lm(mod, typ=1)
print(table)
y_pred = mod.predict(X_test)
mae = mean_absolute_error(y_test, y_pred)
mse = mean_squared_error(y_test, y_pred)
rmse = mean_squared_error(y_test, y_pred, squared = False)
r2 = r2_score(y_test, y_pred)
print("MAE: {}\nMSE: {}\nRMSE: {}\nR_squared: {}".format(mae, mse, rmse, r2))
```

OLS Regression Results

Dep. Variable:	LC50	R-squared:	0.423
Model:	OLS	Adj. R-squared:	0.422
Method:	Least Squares	F-statistic:	531.3
Date:	Wed, 27 Jul 2022	Prob (F-statistic):	1.38e-88
Time:	13:00:21	Log-Likelihood:	-1095.3

No. Observations:	726	AIC:	2195.
Df Residuals:	724	BIC:	2204.
Df Model:	1		
Covariance Type:	nonrobust		

	coef	std err	t	P>|t|	[0.025	0.975]
Intercept	4.0443	0.041	99.405	0.000	3.964	4.124
MLOGP	0.9410	0.041	23.050	0.000	0.861	1.021

Omnibus:	123.874	Durbin-Watson:	2.037
Prob(Omnibus):	0.000	Jarque-Bera (JB):	322.369
Skew:	0.877	Prob(JB):	9.97e-71
Kurtosis:	5.753	Cond. No.	1.04

Notes:
[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.

	df	sum_sq	mean_sq	F	PR(>F)
MLOGP	1.0	637.505102	637.505102	531.280172	1.377389e-88
Residual	724.0	868.757612	1.199941	NaN	NaN

MAE: 0.8431096632604543
MSE: 1.3046098164437467
RMSE: 1.1421951744092367
R_squared: 0.41263278600664965

4 Model Exploration

4.1 Multivariate Model with Every Factor

[26]:
```
mod_1 = smf.ols(formula = "LC50~(CIC0+SM1_Dz+GATS1i+NdsCH+NdssC+MLOGP)", data =␣
↪Train).fit()
print(mod_1.summary())
table = sm.stats.anova_lm(mod_1, typ=1)
print(table)
variables = mod_1.model.exog
y_pred = mod_1.predict(X_test)
mae = mean_absolute_error(y_test, y_pred)
mse = mean_squared_error(y_test, y_pred)
rmse = mean_squared_error(y_test, y_pred, squared = False)
r2 = r2_score(y_test, y_pred)
cls = Linear_Reg_Diagnostic(mod_1)
print("MAE: {}\nMSE: {}\nRMSE: {}\nR_squared: {}".format(mae, mse, rmse, r2))
```

OLS Regression Results

Dep. Variable:	LC50	R-squared:	0.564

Model:	OLS	Adj. R-squared:	0.561
Method:	Least Squares	F-statistic:	155.2
Date:	Wed, 03 Aug 2022	Prob (F-statistic):	4.06e-126
Time:	17:41:47	Log-Likelihood:	-993.50
No. Observations:	726	AIC:	2001.
Df Residuals:	719	BIC:	2033.
Df Model:	6		
Covariance Type:	nonrobust		

	coef	std err	t	P>|t|	[0.025	0.975]
Intercept	4.0379	0.036	113.732	0.000	3.968	4.108
CIC0	0.2863	0.052	5.476	0.000	0.184	0.389
SM1_Dz	0.5260	0.042	12.484	0.000	0.443	0.609
GATS1i	-0.2717	0.045	-5.971	0.000	-0.361	-0.182
NdsCH	0.2326	0.036	6.405	0.000	0.161	0.304
NdssC	0.0476	0.041	1.174	0.241	-0.032	0.127
MLOGP	0.5696	0.055	10.325	0.000	0.461	0.678

Omnibus:	91.056	Durbin-Watson:	1.968
Prob(Omnibus):	0.000	Jarque-Bera (JB):	357.171
Skew:	0.522	Prob(JB):	2.76e-78
Kurtosis:	6.274	Cond. No.	2.91

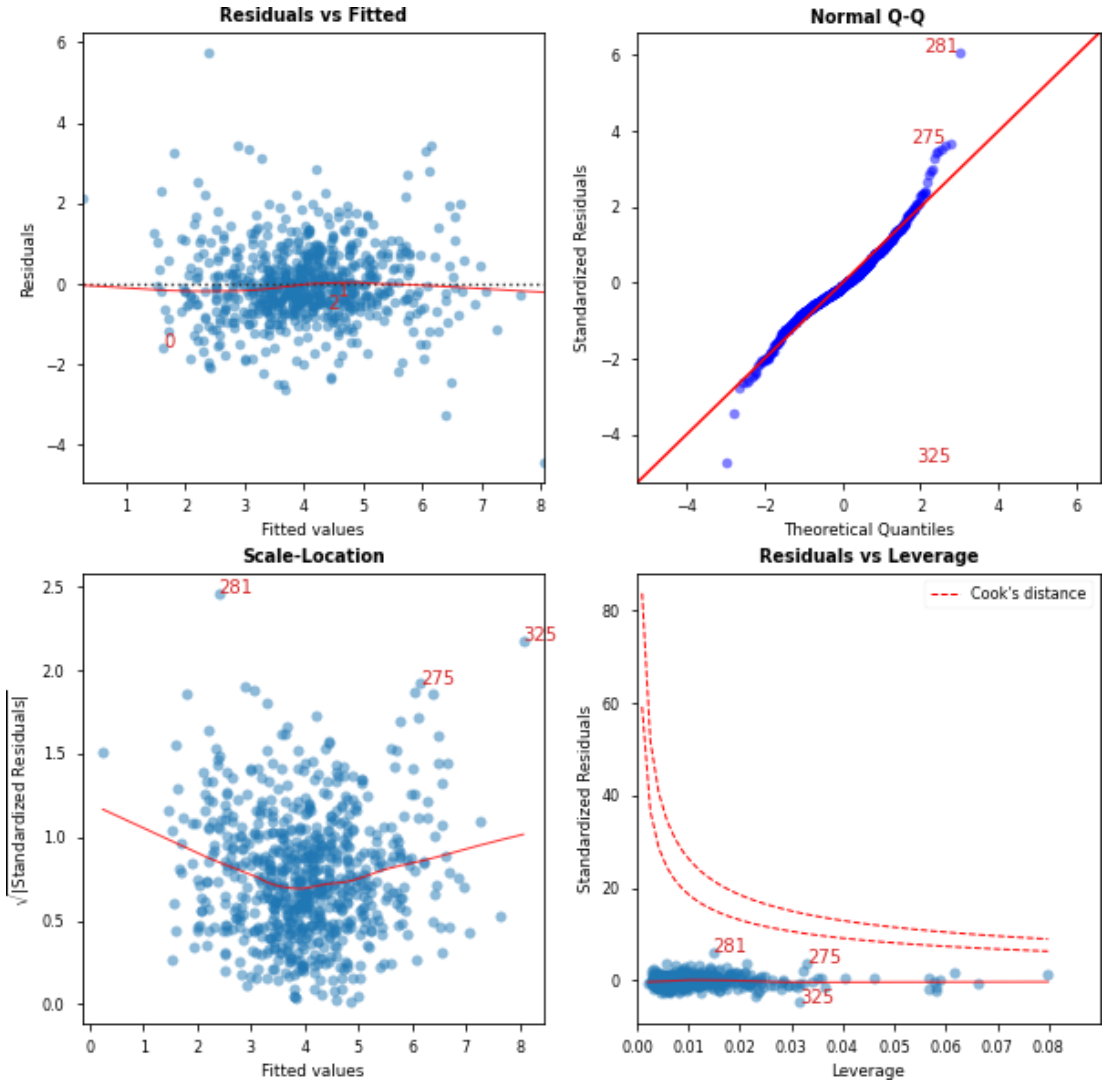
Notes:
[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.

	df	sum_sq	mean_sq	F	PR(>F)
CIC0	1.0	149.824163	149.824163	164.140644	5.401678e-34
SM1_Dz	1.0	332.561060	332.561060	364.339007	5.071243e-66
GATS1i	1.0	222.531248	222.531248	243.795272	1.525815e-47
NdsCH	1.0	46.872834	46.872834	51.351778	1.917520e-12
NdssC	1.0	0.874501	0.874501	0.958064	3.280044e-01
MLOGP	1.0	97.310678	97.310678	106.609222	2.125854e-23
Residual	719.0	656.288230	0.912779	NaN	NaN

MAE: 0.6646693501498407
MSE: 0.8671610192409167
RMSE: 0.9312148083234698
R_squared: 0.609582922391621

[23]:
```
fig, ax = cls()
```

C:\Users\matth\anaconda3\lib\site-packages\statsmodels\graphics\gofplots.py:993: UserWarning: marker is redundantly defined by the 'marker' keyword argument and the fmt string "bo" (-> marker='o'). The keyword argument will take precedence.
  ax.plot(x, y, fmt, **plot_style)



	Features	VIF Factor
0	Intercept	1.00
4	NdsCH	1.11
5	NdssC	1.26
2	SM1_Dz	1.33
3	GATS1i	1.64
1	CIC0	2.22
6	MLOGP	2.40

4.2 Reduced Model

[27]:
```
mod_2 = smf.ols(formula = "LC50~CIC0+SM1_Dz+GATS1i+NdsCH+MLOGP", data = Train).
→fit()
print(mod_2.summary())
table = sm.stats.anova_lm(mod_2, typ=1)
print(table)
table = sm.stats.anova_lm(mod_2,mod_1, typ=1)
print(table)
variables = mod_2.model.exog

y_pred = mod_2.predict(X_test)
mae = mean_absolute_error(y_test, y_pred)
mse = mean_squared_error(y_test, y_pred)
rmse = mean_squared_error(y_test, y_pred, squared = False)
r2 = r2_score(y_test, y_pred)
cls = Linear_Reg_Diagnostic(mod_2)
print("MAE: {}\nMSE: {}\nRMSE: {}\nR_squared: {}".format(mae, mse, rmse, r2))
```

OLS Regression Results

Dep. Variable:	LC50	R-squared:	0.563
Model:	OLS	Adj. R-squared:	0.560
Method:	Least Squares	F-statistic:	185.9
Date:	Wed, 03 Aug 2022	Prob (F-statistic):	5.62e-127
Time:	17:41:50	Log-Likelihood:	-994.20
No. Observations:	726	AIC:	2000.
Df Residuals:	720	BIC:	2028.
Df Model:	5		
Covariance Type:	nonrobust		

	coef	std err	t	P>|t|	[0.025	0.975]
Intercept	4.0381	0.036	113.707	0.000	3.968	4.108
CIC0	0.3066	0.049	6.215	0.000	0.210	0.403
SM1_Dz	0.5407	0.040	13.437	0.000	0.462	0.620
GATS1i	-0.2729	0.045	-5.997	0.000	-0.362	-0.184
NdsCH	0.2432	0.035	6.915	0.000	0.174	0.312
MLOGP	0.5562	0.054	10.302	0.000	0.450	0.662

Omnibus:	92.661	Durbin-Watson:	1.975
Prob(Omnibus):	0.000	Jarque-Bera (JB):	364.204
Skew:	0.533	Prob(JB):	8.20e-80
Kurtosis:	6.302	Cond. No.	2.71

Notes:
[1] Standard Errors assume that the covariance matrix of the errors is correctly

specified.

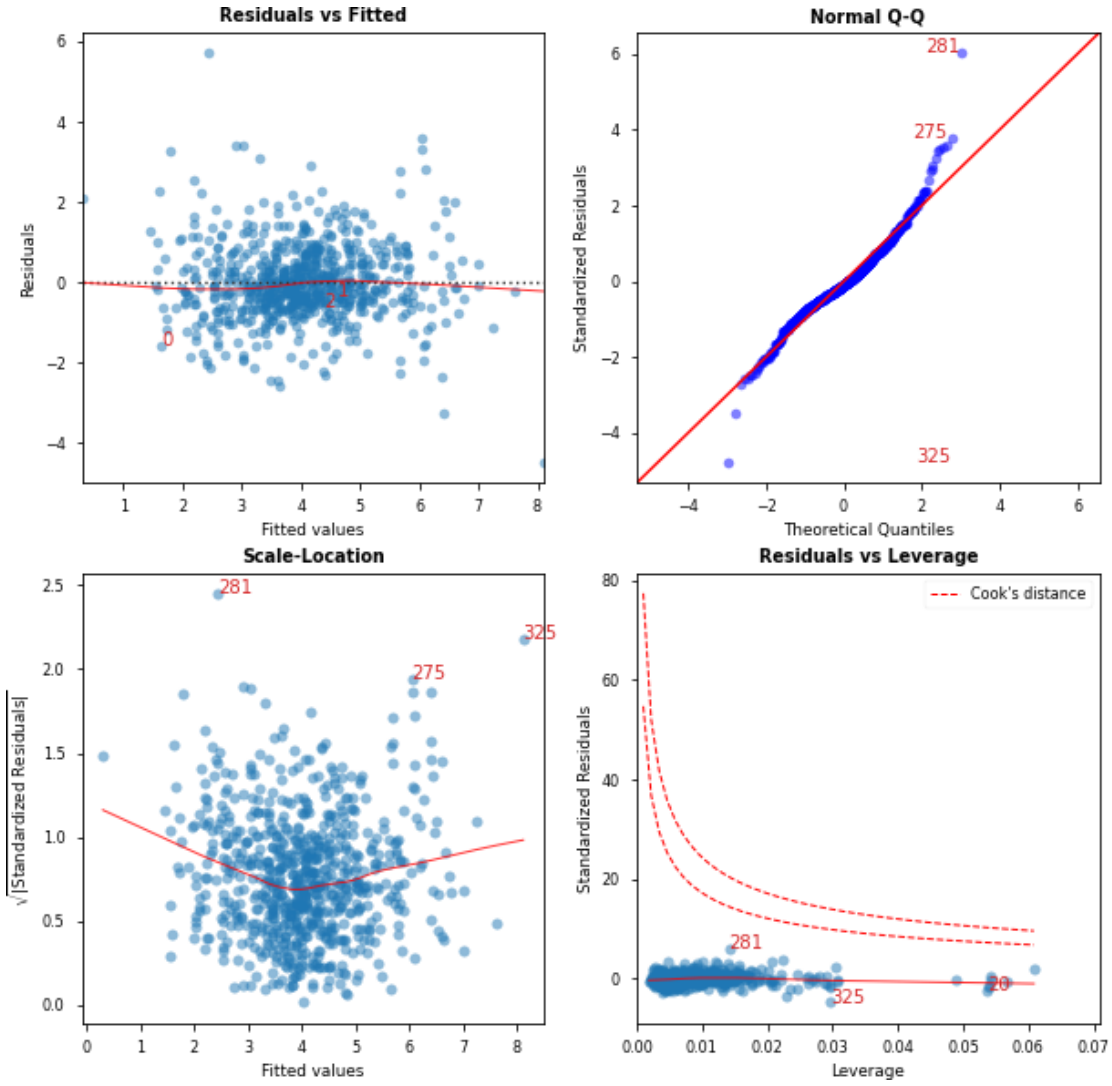
	df	sum_sq	mean_sq	F	PR(>F)
CIC0	1.0	149.824163	149.824163	164.054642	5.539706e-34
SM1_Dz	1.0	332.561060	332.561060	364.148110	5.207671e-66
GATS1i	1.0	222.531248	222.531248	243.667534	1.569730e-47
NdsCH	1.0	46.872834	46.872834	51.324872	1.939865e-12
MLOGP	1.0	96.927875	96.927875	106.134201	2.607435e-23
Residual	720.0	657.545534	0.913258	NaN	NaN

	df_resid	ssr	df_diff	ss_diff	F	Pr(>F)
0	720.0	657.545534	0.0	NaN	NaN	NaN
1	719.0	656.288230	1.0	1.257304	1.377446	0.240925

MAE: 0.6648025405914847
MSE: 0.8725132414740466
RMSE: 0.9340841725851299
R_squared: 0.6071732211751176

[28]:
```
fig, ax = cls()
```

C:\Users\matth\anaconda3\lib\site-packages\statsmodels\graphics\gofplots.py:993: UserWarning: marker is redundantly defined by the 'marker' keyword argument and the fmt string "bo" (-> marker='o'). The keyword argument will take precedence.
  ax.plot(x, y, fmt, **plot_style)



	Features	VIF Factor
0	Intercept	1.00
4	NdsCH	1.04
2	SM1_Dz	1.22
3	GATS1i	1.64
1	CIC0	1.98
5	MLOGP	2.30

5 Interaction Testing

5.1 All First Order Interaction Terms

[29]:
```
mod_3 = smf.ols(formula =
"LC50~(CIC0+SM1_Dz+GATS1i+NdsCH+NdssC+MLOGP)*(CIC0+SM1_Dz+GATS1i+NdsCH+NdssC+MLOGP)",
data = Train).fit()
print(mod_3.summary())
table = sm.stats.anova_lm(mod_3, typ=1)
print(table)
variables = mod_3.model.exog
y_pred = mod_3.predict(X_test)
mae = mean_absolute_error(y_test, y_pred)
mse = mean_squared_error(y_test, y_pred)
rmse = mean_squared_error(y_test, y_pred, squared = False)
r2 = r2_score(y_test, y_pred)
cls = Linear_Reg_Diagnostic(mod_3)
print("MAE: {}\nMSE: {}\nRMSE: {}\nR_squared: {}".format(mae, mse, rmse, r2))
```

OLS Regression Results

Dep. Variable:	LC50	R-squared:	0.614
Model:	OLS	Adj. R-squared:	0.603
Method:	Least Squares	F-statistic:	53.42
Date:	Wed, 03 Aug 2022	Prob (F-statistic):	3.18e-130
Time:	17:42:28	Log-Likelihood:	-949.12
No. Observations:	726	AIC:	1942.
Df Residuals:	704	BIC:	2043.
Df Model:	21		
Covariance Type:	nonrobust		

	coef	std err	t	P>|t|	[0.025	0.975]
Intercept	3.8885	0.049	79.473	0.000	3.792	3.985
CIC0	0.3172	0.060	5.250	0.000	0.199	0.436
SM1_Dz	0.6134	0.044	13.825	0.000	0.526	0.700
GATS1i	-0.2828	0.050	-5.631	0.000	-0.381	-0.184
NdsCH	0.2652	0.048	5.522	0.000	0.171	0.360
NdssC	-0.0940	0.051	-1.836	0.067	-0.194	0.007

MLOGP	0.5200	0.067	7.713	0.000	0.388 0.652
CIC0:SM1_Dz	-0.0868	0.053	-1.647	0.100	-0.190 0.017
CIC0:GATS1i	0.0519	0.048	1.077	0.282	-0.043 0.146
CIC0:NdsCH	-0.0567	0.066	-0.863	0.388	-0.186 0.072
CIC0:NdssC	0.1522	0.052	2.953	0.003	0.051 0.253
CIC0:MLOGP	0.1543	0.034	4.529	0.000	0.087 0.221
SM1_Dz:GATS1i	0.0179	0.050	0.357	0.721	-0.081 0.116
SM1_Dz:NdsCH	0.0152	0.048	0.318	0.751	-0.079 0.109
SM1_Dz:NdssC	0.2140	0.044	4.834	0.000	0.127 0.301
SM1_Dz:MLOGP	-0.0593	0.048	-1.233	0.218	-0.154 0.035
GATS1i:NdsCH	0.0352	0.057	0.618	0.537	-0.077 0.147
GATS1i:NdssC	-0.0024	0.044	-0.054	0.957	-0.089 0.085
GATS1i:MLOGP	-0.0179	0.040	-0.447	0.655	-0.096 0.061
NdsCH:NdssC	0.0499	0.031	1.594	0.111	-0.012 0.111
NdsCH:MLOGP	-0.1503	0.070	-2.159	0.031	-0.287 -0.014
NdssC:MLOGP	-0.0313	0.049	-0.637	0.524	-0.128 0.065

==========

Omnibus:	63.570	Durbin-Watson:	1.943
Prob(Omnibus):	0.000	Jarque-Bera (JB):	249.871
Skew:	0.301	Prob(JB):	5.51e-55
Kurtosis:	5.810	Cond. No.	8.29

==========

Notes:
[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.

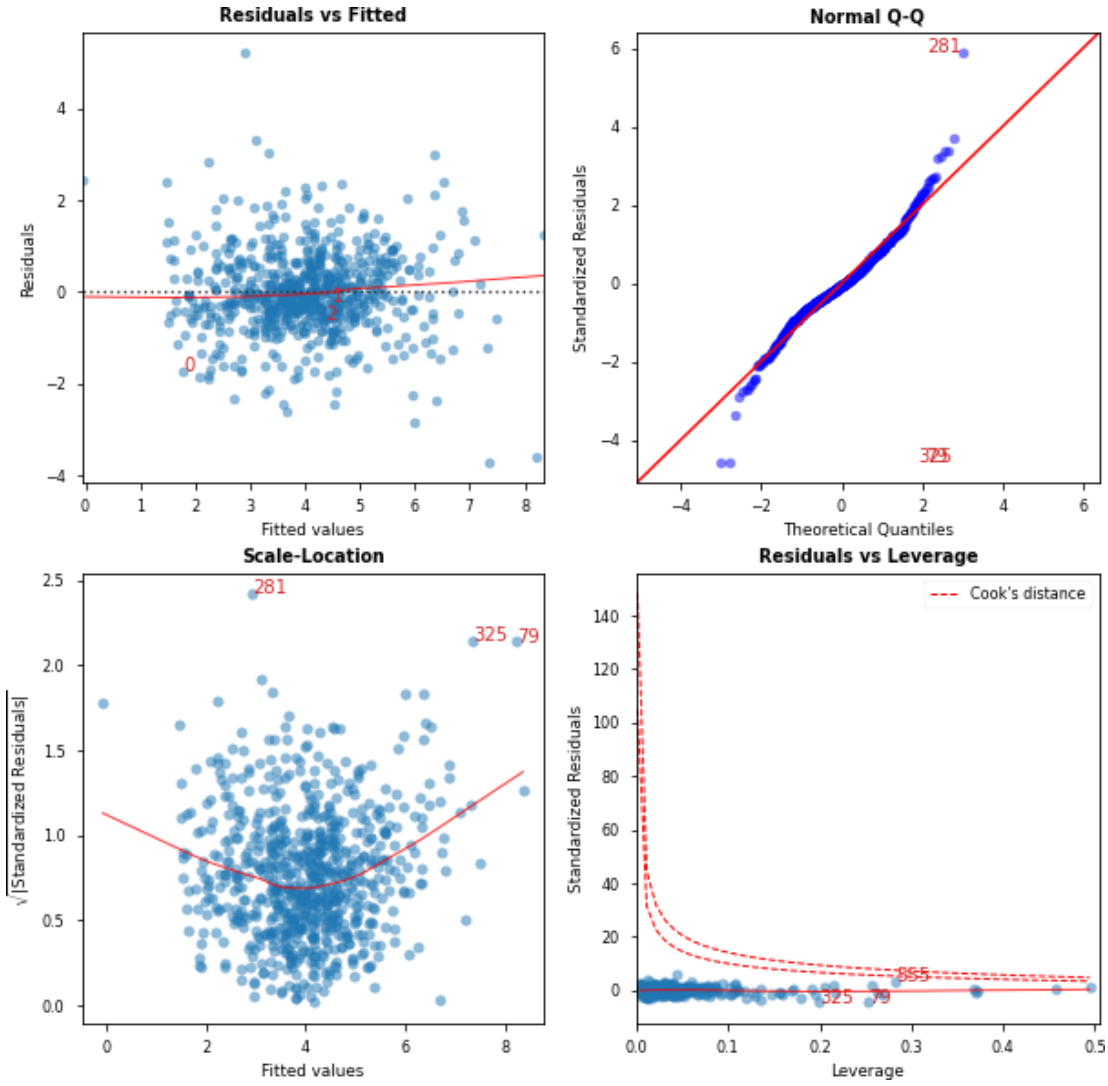
	df	sum_sq	mean_sq	F	PR(>F)
CIC0	1.0	149.824163	149.824163	181.616861	5.428992e-37
SM1_Dz	1.0	332.561060	332.561060	403.130541	3.047164e-71
GATS1i	1.0	222.531248	222.531248	269.752395	1.468387e-51
NdsCH	1.0	46.872834	46.872834	56.819253	1.474241e-13
NdssC	1.0	0.874501	0.874501	1.060070	3.035529e-01

MLOGP	1.0	97.310678	97.310678	117.960011	1.637170e-25
CIC0:SM1_Dz	1.0	0.447692	0.447692	0.542692	4.615646e-01
CIC0:GATS1i	1.0	0.073509	0.073509	0.089107	7.654034e-01
CIC0:NdsCH	1.0	13.029795	13.029795	15.794718	7.788002e-05
CIC0:NdssC	1.0	9.908538	9.908538	12.011131	5.609252e-04
CIC0:MLOGP	1.0	19.988482	19.988482	24.230039	1.065758e-06
SM1_Dz:GATS1i	1.0	1.849727	1.849727	2.242240	1.347346e-01
SM1_Dz:NdsCH	1.0	1.266545	1.266545	1.535306	2.157308e-01
SM1_Dz:NdssC	1.0	16.334674	16.334674	19.800894	9.988010e-06
SM1_Dz:MLOGP	1.0	1.180579	1.180579	1.431098	2.319881e-01
GATS1i:NdsCH	1.0	4.068860	4.068860	4.932272	2.667712e-02
GATS1i:NdssC	1.0	0.008698	0.008698	0.010544	9.182437e-01
GATS1i:MLOGP	1.0	0.153957	0.153957	0.186627	6.658713e-01
NdsCH:NdssC	1.0	2.445800	2.445800	2.964799	8.553430e-02
NdsCH:MLOGP	1.0	4.433940	4.433940	5.374822	2.071503e-02
NdssC:MLOGP	1.0	0.335218	0.335218	0.406351	5.240346e-01
Residual	704.0	580.762216	0.824946	NaN	NaN

MAE: 0.6252053863994669
MSE: 0.8722338398525055
RMSE: 0.9339346014858351
R_squared: 0.6072990146115622

[30]:
```
fig, ax = cls()
```

C:\Users\matth\anaconda3\lib\site-packages\statsmodels\graphics\gofplots.py:993: UserWarning: marker is redundantly defined by the 'marker' keyword argument and the fmt string "bo" (-> marker='o'). The keyword argument will take precedence.
  ax.plot(x, y, fmt, **plot_style)



	Features	VIF Factor
11	CIC0:MLOGP	1.35
14	SM1_Dz:NdssC	1.47
16	GATS1i:NdsCH	1.64
2	SM1_Dz	1.64
17	GATS1i:NdssC	1.68
8	CIC0:GATS1i	1.72
13	SM1_Dz:NdsCH	1.78
18	GATS1i:MLOGP	1.81
12	SM1_Dz:GATS1i	1.93
7	CIC0:SM1_Dz	2.10
0	Intercept	2.11
4	NdsCH	2.16
3	GATS1i	2.21

5	NdssC	2.22
19	NdsCH:NdssC	2.60
15	SM1_Dz:MLOGP	2.71
21	NdssC:MLOGP	2.72
10	CIC0:NdssC	2.99
1	CIC0	3.28
6	MLOGP	3.97
20	NdsCH:MLOGP	5.49
9	CIC0:NdsCH	7.53

5.2 Reduced Interactions Model

[24]:
```
mod_4 = smf.ols(formula = "LC50~CIC0+SM1_Dz+GATS1i+NdsCH+MLOGP+CIC0:NdssC+CIC0:
↪MLOGP+SM1_Dz:NdssC+NdsCH:MLOGP", data = Train).fit()
print(mod_4.summary())
table = sm.stats.anova_lm(mod_4, typ=1)
print(table)
table = sm.stats.anova_lm(mod_4,mod_3, typ=1)
print(table)
variables = mod_4.model.exog
vif = [variance_inflation_factor(variables, i) for i in range(variables.
↪shape[1])]
print(vif)
y_pred = mod_4.predict(X_test)
mae = mean_absolute_error(y_test, y_pred)
mse = mean_squared_error(y_test, y_pred)
rmse = mean_squared_error(y_test, y_pred, squared = False)
r2 = r2_score(y_test, y_pred)
cls = Linear_Reg_Diagnostic(mod_4)
print("MAE: {}\nMSE: {}\nRMSE: {}\nR_squared: {}".format(mae, mse, rmse, r2))
```

OLS Regression Results

Dep. Variable:	LC50	R-squared:	0.603
Model:	OLS	Adj. R-squared:	0.598
Method:	Least Squares	F-statistic:	121.0
Date:	Wed, 27 Jul 2022	Prob (F-statistic):	2.44e-137
Time:	13:00:21	Log-Likelihood:	-959.51
No. Observations:	726	AIC:	1939.
Df Residuals:	716	BIC:	1985.
Df Model:	9		
Covariance Type:	nonrobust		

	coef	std err	t	P>|t|	[0.025	0.975]
Intercept	3.9337	0.038	103.544	0.000	3.859	4.008
CIC0	0.3073	0.049	6.314	0.000	0.212	0.403
SM1_Dz	0.5587	0.040	14.049	0.000	0.481	0.637

GATS1i	-0.2886	0.044	-6.526	0.000	-0.375	-0.202
NdsCH	0.2483	0.036	6.813	0.000	0.177	0.320
MLOGP	0.5476	0.053	10.290	0.000	0.443	0.652
CIC0:NdssC	0.1103	0.035	3.192	0.001	0.042	0.178
CIC0:MLOGP	0.1440	0.031	4.578	0.000	0.082	0.206
SM1_Dz:NdssC	0.1658	0.038	4.370	0.000	0.091	0.240
NdsCH:MLOGP	-0.1739	0.034	-5.123	0.000	-0.241	-0.107

Omnibus:	61.426	Durbin-Watson:	1.938
Prob(Omnibus):	0.000	Jarque-Bera (JB):	266.636
Skew:	0.229	Prob(JB):	1.26e-58
Kurtosis:	5.933	Cond. No.	3.63

Notes:
[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.

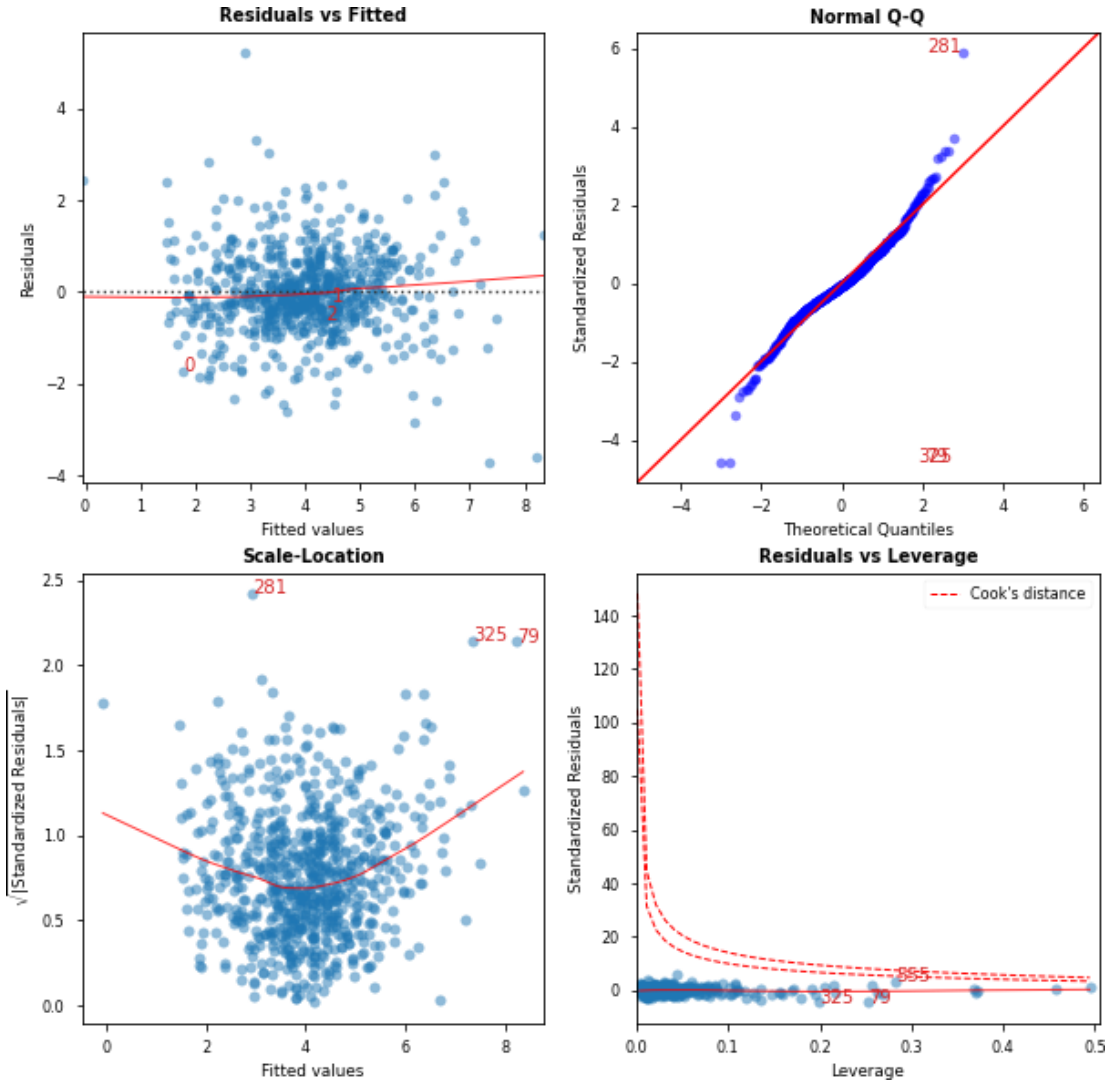
	df	sum_sq	mean_sq	F	PR(>F)
CIC0	1.0	149.824163	149.824163	179.503126	1.095941e-36
SM1_Dz	1.0	332.561060	332.561060	398.438735	8.115997e-71
GATS1i	1.0	222.531248	222.531248	266.612901	3.482681e-51
NdsCH	1.0	46.872834	46.872834	56.157966	1.977608e-13
MLOGP	1.0	96.927875	96.927875	116.128508	3.380675e-25
CIC0:NdssC	1.0	2.973956	2.973956	3.563073	5.948244e-02
CIC0:MLOGP	1.0	17.586136	17.586136	21.069809	5.229643e-06
SM1_Dz:NdssC	1.0	17.462449	17.462449	20.921620	5.637720e-06
NdsCH:MLOGP	1.0	21.906098	21.906098	26.245520	3.870201e-07
Residual	716.0	597.616893	0.834660	NaN	NaN

	df_resid	ssr	df_diff	ss_diff	F	Pr(>F)
0	716.0	597.616893	0.0	NaN	NaN	NaN
1	704.0	580.762216	12.0	16.854678	1.702603	0.061921

[1.255416809499518, 2.1044981905482576, 1.3004538978884799, 1.6934312959727862, 1.2264784684887298, 2.443022742239267, 1.3292692131540762, 1.1348823438792845, 1.0676744718552331, 1.2905891804746232]
MAE: 0.6466933021246712
MSE: 0.9118498576541835
RMSE: 0.9549082980339962
R_squared: 0.5894629154864519

[31]:
```
fig, ax = cls()
```

C:\Users\matth\anaconda3\lib\site-packages\statsmodels\graphics\gofplots.py:993: UserWarning: marker is redundantly defined by the 'marker' keyword argument and the fmt string "bo" (-> marker='o'). The keyword argument will take precedence.
  ax.plot(x, y, fmt, **plot_style)



	Features	VIF Factor
11	CIC0:MLOGP	1.35
14	SM1_Dz:NdssC	1.47
16	GATS1i:NdsCH	1.64
2	SM1_Dz	1.64
17	GATS1i:NdssC	1.68
8	CIC0:GATS1i	1.72
13	SM1_Dz:NdsCH	1.78
18	GATS1i:MLOGP	1.81
12	SM1_Dz:GATS1i	1.93
7	CIC0:SM1_Dz	2.10
0	Intercept	2.11
4	NdsCH	2.16
3	GATS1i	2.21

5	NdssC	2.22
19	NdsCH:NdssC	2.60
15	SM1_Dz:MLOGP	2.71
21	NdssC:MLOGP	2.72
10	CIC0:NdssC	2.99
1	CIC0	3.28
6	MLOGP	3.97
20	NdsCH:MLOGP	5.49
9	CIC0:NdsCH	7.53

6 Evaluation of Linear Models

There doesn't seem to be much good performance with Linear Models on this dataset. While the model with all first order interaction terms did perform the best, it may be due to there being more features. There was no significance between both reduced models and their respective parent model. The models with interaciton terms had less MAE, but had worse MSE, meaning errors may be bigger when interaction terms are included. R^2 on the test data for the models without interaction was higher as well. It is unlikely that interaction is present. Assumptions for linear regression hold for each model.

7 KNN Regression

7.1 KNN Example 6 Neighbors

[32]:
```
knn1 = KNeighborsRegressor(n_neighbors=6)
knn1.fit(X_train, y_train)
knn1.score(X_test, y_test)
y_pred = knn1.predict(X_test)
mae = mean_absolute_error(y_test, y_pred)
mse = mean_squared_error(y_test, y_pred)
rmse = mean_squared_error(y_test, y_pred, squared = False)
r2 = r2_score(y_test, y_pred)
print("MAE: {}\nMSE: {}\nRMSE: {}\nR_squared: {}".format(mae, mse, rmse, r2))
```

```
MAE: 0.5972747252747254
MSE: 0.7172062567155068
RMSE: 0.8468803083762821
R_squared: 0.6770962202216795
```

7.2 Grid Search Optimization for KNN

[33]:
```
scoring = {"MSE": "neg_mean_squared_error", "R2":"r2"}

# Setting refit='AUC', refits an estimator on the whole dataset with the
# parameter setting that has the best cross-validated AUC score.
# That estimator is made available at ``gs.best_estimator_`` along with
# parameters like ``gs.best_score_``, ``gs.best_params_`` and
```

```
# ``gs.best_index_``
gs = GridSearchCV(
    KNeighborsRegressor(),
    param_grid={"n_neighbors": range(1, 6), "weights":["uniform","distance"],
↪"p":[1,2]},
    scoring=scoring,
    refit="MSE",
    return_train_score=True,
    n_jobs = -1,
    cv = 100,
    verbose = 3
)
gs.fit(X_train, y_train)
results = gs.cv_results_
```

Fitting 100 folds for each of 20 candidates, totalling 2000 fits

[34]:
```
gs.best_params_
```

[34]: {'n_neighbors': 5, 'p': 1, 'weights': 'distance'}

[35]:
```
knn = gs.best_estimator_
```

[36]:
```
y_pred = knn.predict(X_test)
mae = mean_absolute_error(y_test, y_pred)
mse = mean_squared_error(y_test, y_pred)
rmse = mean_squared_error(y_test, y_pred, squared = False)
r2 = r2_score(y_test, y_pred)
print("MAE: {}\nMSE: {}\nRMSE: {}\nR_squared: {}".format(mae, mse, rmse, r2))
```

MAE: 0.549976177393366
MSE: 0.6426008012295652
RMSE: 0.8016238527074685
R_squared: 0.7106854190650043

7.3 PCA and KNN

[28]:
```
def get_models():
    models = dict()
    for i in range(1,7):
        steps = [('pca', PCA(n_components=i)), ('m',
↪KNeighborsRegressor(n_neighbors=5, p = 1, weights = "distance"))]
        models[str(i)] = Pipeline(steps=steps)
    return models

# evaluate a given model using cross-validation
def evaluate_model(model, X, y):
    cv = KFold(n_splits=5, random_state=101, shuffle = True)
```

```
    scores = cross_val_score(model, X, y, scoring='neg_mean_squared_error',␣
↪cv=cv, n_jobs=-1, error_score='raise')
    return scores
# get the models to evaluate
models = get_models()
# evaluate the models and store results
results, names = list(), list()
for name, model in models.items():
    scores = evaluate_model(model, X_train, y_train)
    results.append(scores)
    names.append(name)
    print('>%s %.3f (%.3f)' % (name, mean(scores), std(scores)))
```

```
>1 -1.738 (0.114)
>2 -1.161 (0.140)
>3 -0.950 (0.105)
>4 -0.812 (0.157)
>5 -0.866 (0.141)
>6 -0.847 (0.148)
```

[29]:
```
steps = [('pca', PCA(n_components=4)), ('m', KNeighborsRegressor(n_neighbors=5,␣
↪p = 1, weights = "distance"))]
model = Pipeline(steps=steps)
model.fit(X_train, y_train)
y_pred = model.predict(X_test)
mae = mean_absolute_error(y_test, y_pred)
mse = mean_squared_error(y_test, y_pred)
rmse = mean_squared_error(y_test, y_pred, squared = False)
r2 = r2_score(y_test, y_pred)
print("MAE: {}\nMSE: {}\nRMSE: {}\nR_squared: {}".format(mae, mse, rmse, r2))
```

```
MAE: 0.5937420511872225
MSE: 0.7569008399616851
RMSE: 0.8700004827364667
R_squared: 0.6592247490139198
```

8 Evaluation of KNN

KNN improves performance on the Test dataset. It was optimized, and led to a higher R^2 score and lower *MSE*. PCA compresses the data too much and does not allow for meaningful relationships to be seen by the model. KNN with 5 neighbors, l1 distance performs the best on the data.

9 Gradient Descent

[30]:
```
sgd_reg = SGDRegressor(max_iter=1000, tol=1e-3, penalty="l2", eta0=0.01)
sgd_reg.fit(X_train,    y_train.values.ravel())
sgd_reg.intercept_, sgd_reg.coef_
y_pred = sgd_reg.predict(X_test)
mae = mean_absolute_error(y_test, y_pred)
mse = mean_squared_error(y_test, y_pred)
rmse = mean_squared_error(y_test, y_pred, squared = False)
r2 = r2_score(y_test, y_pred)
print("MAE: {}\nMSE: {}\nRMSE: {}\nR_squared: {}".format(mae, mse, rmse, r2))
```

```
MAE: 0.6641009021334089
MSE: 0.8673091952076947
RMSE: 0.931294365497663
R_squared: 0.6095162099510966
```

10 Evaluation of Gradient Descent

Same performance as linear model but is outperformed by KNN. Cant be optimized as well since parameters are continuous

11 11. Ridge, Lasso, and Elasticnet Regularization

11.1 Ridge Regression

[39]:
```
poly = PolynomialFeatures(2)
poly_X_train = poly.fit_transform(X_train)
poly_X_test = poly.fit_transform(X_test)
ridge_reg = Ridge(alpha=1 , solver= "auto")
ridge_reg.fit(poly_X_train , y_train)
y_pred= ridge_reg.predict(poly_X_test)
mae = mean_absolute_error(y_test, y_pred)
mse = mean_squared_error(y_test, y_pred)
rmse = mean_squared_error(y_test, y_pred, squared = False)
r2 = r2_score(y_test, y_pred)
print("MAE: {}\nMSE: {}\nRMSE: {}\nR_squared: {}".format(mae, mse, rmse, r2))
```

```
MAE: 0.6192753750697797
MSE: 0.8440158736842909
RMSE: 0.9187033654473521
R_squared: 0.6200034324105659
```

[62]:
```
intercept = ridge_reg.intercept_[0]
print("E(y) = " + str(intercept), end =" ")
i = 1
for coef in ridge_reg.coef_[0]:
    print("+ " + str(coef) + "*X_"+ str(i),end =" ")
```

```
    i+=1
```

E(y) = 3.892252012873493 + 0.0*X_1 + 0.3009287547124961*X_2 + 0.5846542581855313*X_3 + -0.2885722891861736*X_4 + 0.33561044270534446*X_5 + -0.04691621569339529*X_6 + 0.5147181910314026*X_7 + -0.011844581261028983*X_8 + -0.07988945669958426*X_9 + 0.0724036613395784*X_10 + -0.027615552497574667*X_11 + 0.17945067847378657*X_12 + 0.16240783769337988*X_13 + 0.04029647132214782*X_14 + 0.017151044825168892*X_15 + -0.005330546352793698*X_16 + 0.23548063628815477*X_17 + -0.07834081616696212*X_18 + 0.009090663384291042*X_19 + 0.014132939074507442*X_20 + -0.02385617257690637*X_21 + -0.013281152434592177*X_22 + -0.02893142065627501*X_23 + 0.06865771394750197*X_24 + -0.16582416385791565*X_25 + -0.0317079979197357*X_26 + -0.059135522124548326*X_27 + 0.0019317900535462068*X_28

11.2 Lasso Regression

[65]:
```
lasso_reg = Lasso(alpha = 0.05)
lasso_reg.fit(poly_X_train,    y_train)
y_pred = lasso_reg.predict(poly_X_test)
mae = mean_absolute_error(y_test, y_pred)
mse = mean_squared_error(y_test, y_pred)
rmse = mean_squared_error(y_test, y_pred, squared = False)
r2 = r2_score(y_test, y_pred)
print("MAE: {}\nMSE: {}\nRMSE: {}\nR_squared: {}".format(mae, mse, rmse, r2))
```

MAE: 0.6531654769265813
MSE: 0.8695790137885683
RMSE: 0.9325122057048735
R_squared: 0.608494282169078

[67]:
```
intercept = lasso_reg.intercept_[0]
print("E(y) = " + str(intercept), end =" ")
i = 1
for coef in lasso_reg.coef_:
    print("+ " + str(coef) + "*X_"+ str(i),end =" ")
    i+=1
```

E(y) = 3.9173103213192504 + 0.0*X_1 + 0.19886736243419062*X_2 + 0.4559465671701525*X_3 + -0.21256611454121532*X_4 + 0.18174835873080378*X_5 + 0.0*X_6 + 0.5931181518839378*X_7 + 0.04905532044355343*X_8 + -0.0*X_9 + 0.0*X_10 + -0.0*X_11 + 0.05023123430594591 4*X_12 + 0.04983652536927835*X_13 + 0.02268194583406715*X_14 + 0.0*X_15 + 0.0*X_16 + 0.09999300518796372*X_17 + -0.03501765577178401*X_18 + -0.0*X_19 + 0.0*X_20 + 0.0*X_21 + 0.0*X_22 + 0.0*X_23 + 0.017039571692934158*X_24 + -0.10798183174529531*X_25 + 0.011519941933311422*X_26 + 0.0*X_27 + 0.0*X_28

11.3 Elasticnet

[68]:
```
elastic_net = ElasticNet(alpha=0.01, l1_ratio=0.01)
elastic_net.fit(poly_X_train, y_train)
y_pred = elastic_net.predict(poly_X_test)
mae = mean_absolute_error(y_test, y_pred)
mse = mean_squared_error(y_test, y_pred)
rmse = mean_squared_error(y_test, y_pred, squared = False)
r2 = r2_score(y_test, y_pred)
print("MAE: {}\nMSE: {}\nRMSE: {}\nR_squared: {}".format(mae, mse, rmse, r2))
```

```
MAE: 0.6197670116899672
MSE: 0.8445688324608495
RMSE: 0.9190042613942818
R_squared: 0.6197544768592995
```

[69]:
```
intercept = elastic_net.intercept_[0]
print("E(y) = " + str(intercept), end =" ")
i = 1
for coef in elastic_net.coef_:
    print("+ " + str(coef) + "*X_"+ str(i),end =" ")
    i+=1
```

```
E(y) = 3.8846309242680777 + 0.0*X_1 + 0.2927933522346351*X_2 +
0.5701587565393743*X_3 + -0.285315529613328*X_4 + 0.3232608010829245*X_5 +
-0.040683191849905674*X_6 + 0.5176625959975297*X_7 + -0.005686391706115492*X_8 +
-0.07347373561585265*X_9 + 0.06744551223755198*X_10 + -0.026991311089282124*X_11
+ 0.17140210905060943*X_12 + 0.1489001291691723*X_13 + 0.04387944907565253*X_14
+ 0.015432694442330381*X_15 + -0.003522939339889553*X_16 +
0.22665360496982379*X_17 + -0.07987184434323709*X_18 + 0.010778170793633652*X_19
+ 0.015419767927768628*X_20 + -0.019190216994567962*X_21 +
-0.004776018749484832*X_22 + -0.026267038845699857*X_23 +
0.06587236709111102*X_24 + -0.16512229677517617*X_25 +
-0.029005690231739913*X_26 + -0.05148489107405104*X_27 +
0.008550690699392257*X_28
```

12 Evaluation of Regularization

Slight improvement, but not as good as KNN

13 Ridge, Lasso, and Elasticnet Regularization

[74]:
```
scoring = {"MSE": "neg_mean_squared_error", "R2":"r2"}

# Setting refit='AUC', refits an estimator on the whole dataset with the
# parameter setting that has the best cross-validated AUC score.
# That estimator is made available at ``gs.best_estimator_`` along with
# parameters like ``gs.best_score_``, ``gs.best_params_`` and
```

```
# ``gs.best_index_``
gs = GridSearchCV(
    RandomForestRegressor(),
    param_grid={"n_estimators": [10,25,30,50,100,200], "max_depth":
↪[2,3,5,10,20], "min_samples_leaf":[5,10,20,50,100,200]},
    scoring=scoring,
    refit="MSE",
    return_train_score=True,
    n_jobs = -1,
    cv = 20,
    verbose = 3
)
gs.fit(X_train, y_train.values.ravel())
results = gs.cv_results_
```

Fitting 20 folds for each of 180 candidates, totalling 3600 fits

[75]:
```
gs.best_params_
```

[75]: {'max_depth': 20, 'min_samples_leaf': 5, 'n_estimators': 50}

[76]:
```
rf = gs.best_estimator_
```

[77]:
```
y_pred = rf.predict(X_test)
mae = mean_absolute_error(y_test, y_pred)
mse = mean_squared_error(y_test, y_pred)
rmse = mean_squared_error(y_test, y_pred, squared = False)
r2 = r2_score(y_test, y_pred)
print("MAE: {}\nMSE: {}\nRMSE: {}\nR_squared: {}".format(mae, mse, rmse, r2))
```

MAE: 0.6061146426280116
MSE: 0.7715916605381403
RMSE: 0.8784029033069849
R_squared: 0.6526105826597295

14 Evaluation of Random Forest Regression

Allows for the learning of nonlinear trends, increases in performance compared to linear models and regularization techniques, indicating there is a nonlinear trend in the data.

15 Neural Network Regression

[268]:
```
NN_model = Sequential()

NN_model.add(Dense(64, kernel_initializer='normal',input_dim = 6,␣
↪activation='relu'))
```

```
NN_model.add(Dense(64, kernel_initializer='normal',activation='tanh'))
NN_model.add(Dense(128, kernel_initializer='normal',activation='tanh'))
NN_model.add(Dense(64, kernel_initializer='normal',activation='tanh'))
NN_model.add(Dense(32, kernel_initializer='normal',activation='tanh'))

NN_model.add(Dense(1, kernel_initializer='normal',activation='linear'))

NN_model.compile(loss='mean_squared_error', optimizer='rmsprop',␣
↪metrics=['mean_squared_error'])
NN_model.summary()
```

```
Model: "sequential_32"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_167 (Dense)            (None, 64)                448

dense_168 (Dense)            (None, 64)                4160

dense_169 (Dense)            (None, 128)               8320

dense_170 (Dense)            (None, 64)                8256

dense_171 (Dense)            (None, 32)                2080

dense_172 (Dense)            (None, 1)                 33

=================================================================
Total params: 23,297
Trainable params: 23,297
Non-trainable params: 0
_________________________________________________________________
```

[269]:
```
checkpoint_name = 'Weights-{epoch:03d}--{val_loss:.5f}.hdf5'
checkpoint = ModelCheckpoint(checkpoint_name, monitor='val_loss', verbose = 3,␣
↪save_best_only = True, mode ='auto')
callbacks_list = [checkpoint]
```

[270]:
```
with tf.device("/CPU:0"):
    NN_model.fit(X_train, y_train, epochs=100, batch_size=1, validation_split =␣
↪0.2, callbacks=callbacks_list,verbose = 0)
```

```
Epoch 1: val_loss improved from inf to 0.93916, saving model to Weights-001--
0.93916.hdf5

Epoch 2: val_loss improved from 0.93916 to 0.88233, saving model to Weights-002
--0.88233.hdf5
```

Epoch 3: val_loss did not improve from 0.88233

Epoch 4: val_loss improved from 0.88233 to 0.84274, saving model to Weights-004--0.84274.hdf5

Epoch 5: val_loss did not improve from 0.84274

Epoch 6: val_loss improved from 0.84274 to 0.82479, saving model to Weights-006--0.82479.hdf5

Epoch 7: val_loss did not improve from 0.82479

Epoch 8: val_loss did not improve from 0.82479

Epoch 9: val_loss did not improve from 0.82479

Epoch 10: val_loss improved from 0.82479 to 0.78585, saving model to Weights-010--0.78585.hdf5

Epoch 11: val_loss improved from 0.78585 to 0.76202, saving model to Weights-011--0.76202.hdf5

Epoch 12: val_loss did not improve from 0.76202

Epoch 13: val_loss did not improve from 0.76202

Epoch 14: val_loss improved from 0.76202 to 0.75242, saving model to Weights-014--0.75242.hdf5

Epoch 15: val_loss did not improve from 0.75242

Epoch 16: val_loss did not improve from 0.75242

Epoch 17: val_loss did not improve from 0.75242

Epoch 18: val_loss improved from 0.75242 to 0.74493, saving model to Weights-018--0.74493.hdf5

Epoch 19: val_loss did not improve from 0.74493

Epoch 20: val_loss did not improve from 0.74493

Epoch 21: val_loss did not improve from 0.74493

Epoch 22: val_loss improved from 0.74493 to 0.73038, saving model to Weights-022--0.73038.hdf5

Epoch 23: val_loss did not improve from 0.73038

Epoch 24: val_loss did not improve from 0.73038

Epoch 25: val_loss did not improve from 0.73038

Epoch 26: val_loss did not improve from 0.73038

Epoch 27: val_loss did not improve from 0.73038

Epoch 28: val_loss did not improve from 0.73038

Epoch 29: val_loss did not improve from 0.73038

Epoch 30: val_loss did not improve from 0.73038

Epoch 31: val_loss did not improve from 0.73038

Epoch 32: val_loss did not improve from 0.73038

Epoch 33: val_loss did not improve from 0.73038

Epoch 34: val_loss did not improve from 0.73038

Epoch 35: val_loss did not improve from 0.73038

Epoch 36: val_loss did not improve from 0.73038

Epoch 37: val_loss did not improve from 0.73038

Epoch 38: val_loss did not improve from 0.73038

Epoch 39: val_loss did not improve from 0.73038

Epoch 40: val_loss improved from 0.73038 to 0.71186, saving model to Weights-040 --0.71186.hdf5

Epoch 41: val_loss did not improve from 0.71186

Epoch 42: val_loss did not improve from 0.71186

Epoch 43: val_loss did not improve from 0.71186

Epoch 44: val_loss did not improve from 0.71186

Epoch 45: val_loss did not improve from 0.71186

Epoch 46: val_loss did not improve from 0.71186

Epoch 47: val_loss did not improve from 0.71186

Epoch 48: val_loss did not improve from 0.71186

Epoch 49: val_loss did not improve from 0.71186

Epoch 50: val_loss did not improve from 0.71186

Epoch 51: val_loss did not improve from 0.71186

Epoch 52: val_loss did not improve from 0.71186

Epoch 53: val_loss did not improve from 0.71186

Epoch 54: val_loss did not improve from 0.71186

Epoch 55: val_loss did not improve from 0.71186

Epoch 56: val_loss did not improve from 0.71186

Epoch 57: val_loss did not improve from 0.71186

Epoch 58: val_loss did not improve from 0.71186

Epoch 59: val_loss did not improve from 0.71186

Epoch 60: val_loss did not improve from 0.71186

Epoch 61: val_loss did not improve from 0.71186

Epoch 62: val_loss did not improve from 0.71186

Epoch 63: val_loss did not improve from 0.71186

Epoch 64: val_loss did not improve from 0.71186

Epoch 65: val_loss did not improve from 0.71186

Epoch 66: val_loss did not improve from 0.71186

Epoch 67: val_loss did not improve from 0.71186

Epoch 68: val_loss did not improve from 0.71186

Epoch 69: val_loss did not improve from 0.71186

Epoch 70: val_loss did not improve from 0.71186

Epoch 71: val_loss did not improve from 0.71186

Epoch 72: val_loss did not improve from 0.71186

Epoch 73: val_loss did not improve from 0.71186

Epoch 74: val_loss did not improve from 0.71186

Epoch 75: val_loss did not improve from 0.71186

Epoch 76: val_loss did not improve from 0.71186

Epoch 77: val_loss did not improve from 0.71186

Epoch 78: val_loss did not improve from 0.71186

Epoch 79: val_loss did not improve from 0.71186

Epoch 80: val_loss did not improve from 0.71186

Epoch 81: val_loss did not improve from 0.71186

Epoch 82: val_loss did not improve from 0.71186

Epoch 83: val_loss did not improve from 0.71186

Epoch 84: val_loss did not improve from 0.71186

Epoch 85: val_loss did not improve from 0.71186

Epoch 86: val_loss did not improve from 0.71186

Epoch 87: val_loss did not improve from 0.71186

Epoch 88: val_loss did not improve from 0.71186

Epoch 89: val_loss did not improve from 0.71186

Epoch 90: val_loss did not improve from 0.71186

Epoch 91: val_loss did not improve from 0.71186

Epoch 92: val_loss did not improve from 0.71186

Epoch 93: val_loss did not improve from 0.71186

Epoch 94: val_loss did not improve from 0.71186

Epoch 95: val_loss did not improve from 0.71186

Epoch 96: val_loss did not improve from 0.71186

Epoch 97: val_loss did not improve from 0.71186

Epoch 98: val_loss did not improve from 0.71186

Epoch 99: val_loss did not improve from 0.71186

Epoch 100: val_loss did not improve from 0.71186

[272]:
```
weights_file = "Weights-040--0.71186.hdf5" # choose the best checkpoint
NN_model.load_weights(weights_file) # load it
NN_model.compile(loss="mean_squared_error", optimizer="adam",␣
↪metrics=["mean_squared_error"])
```

[273]:
```
y_pred = NN_model.predict(X_test)
mae = mean_absolute_error(y_test, y_pred)
mse = mean_squared_error(y_test, y_pred)
rmse = mean_squared_error(y_test, y_pred, squared = False)
r2 = r2_score(y_test, y_pred)
print("MAE: {}\nMSE: {}\nRMSE: {}\nR_squared: {}".format(mae, mse, rmse, r2))
```

```
6/6 [==============================] - 0s 3ms/step
MAE: 0.5881101118706086
MSE: 0.7509005102090602
RMSE: 0.8665451576283029
R_squared: 0.6619262440704632
```

16 Evaluation of Neural Network

Took a while to train, but negligible increase in performance compared to the random forest and worse performance than KNN. Distance based modeling is a neccessity in this case.