Fathead Minnow Toxicity Analysis

Yushuo (Shawn) Han

May 2020

1 Introduction

Fathead minnow (*Pimephales promelas*) is a species of freshwater fish from the Cyprinid family and can often be found in North American waters. Since fathead minnow is tolerant of harsh aquatic environments, it is often used to analyze various chemicals' and toxins' effects on aquatic organisms.

A research conducted by M. Cassotti et al. (DOI: 10.1080/1062936X.2015.1018938), published on *SAR and QSAR in Environmental Research*, derived six moleculer descriptors that are used to predict the concentration of LC_{50}, an acute toxin, in fathead minnow. The dataset collected consists of 908 instances, and 726 instances were used to train the model in the research.

In the following, the correlation between the six molecular descriptors and toxicity (LC_{50}) is analyzed. In particular, an Support Vector Regression (SVR) model will be developed to predict the toxicity level in fathead minnow, given the concentration of the six molecular descriptors, using Python Scikit-learn and Tensorflow with Keras.

2 Data Cleaning

[1]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
%matplotlib inline
```

Read data from the CSV file. Note that the dataset is obtained from the UCI Machine Learning Repository.

[2]:
```
df = pd.read_csv('qsar_fish_toxicity.csv')
```

[3]:
```
df.info()
df.tail()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 907 entries, 0 to 906
Data columns (total 1 columns):
 #   Column                            Non-Null Count  Dtype 
---  ------                            --------------  ----- 
 0   3.26;0.829;1.676;0;1;1.453;3.770  907 non-null    object
dtypes: object(1)
memory usage: 7.2+ KB
```

[3]:
```
       3.26;0.829;1.676;0;1;1.453;3.770
902  2.801;0.728;2.226;0;2;0.736;3.109
903  3.652;0.872;0.867;2;3;3.983;4.040
904  3.763;0.916;0.878;0;6;2.918;4.818
905  2.831;1.393;1.077;0;1;0.906;5.317
906  4.057;1.032;1.183;1;3;4.754;8.201
```

It is shown that the data must be parsed and columns must be manually created.

[4]:
```
df2 = pd.DataFrame({"3.26;0.829;1.676;0;1;1.453;3.770":['3.26;0.829;1.676;0;1;1.
↪453;3.770']})
df = pd.concat([df, df2], ignore_index=True)
df.tail()
```

[4]:
```
       3.26;0.829;1.676;0;1;1.453;3.770
903  3.652;0.872;0.867;2;3;3.983;4.040
904  3.763;0.916;0.878;0;6;2.918;4.818
905  2.831;1.393;1.077;0;1;0.906;5.317
906  4.057;1.032;1.183;1;3;4.754;8.201
907   3.26;0.829;1.676;0;1;1.453;3.770
```

[5]:
```
df.columns = ['Chemicals']
df['Chemicals'].iloc[0].split(";")
df['CIC0'] = df['Chemicals'].apply(lambda x: float(x.split(";")[0]))
df['SM1_Dz(Z)'] = df['Chemicals'].apply(lambda x: float(x.split(";")[1]))
df['GATS1i'] = df['Chemicals'].apply(lambda x: float(x.split(";")[2]))
df['NdsCH'] = df['Chemicals'].apply(lambda x: float(x.split(";")[3]))
df['NdssC'] = df['Chemicals'].apply(lambda x: float(x.split(";")[4]))
df['MLOGP'] = df['Chemicals'].apply(lambda x: float(x.split(";")[5]))
df['LC50 [-LOG(mol/L)]'] = df['Chemicals'].apply(lambda x: float(x.split(";
↪")[-1]))
df.drop('Chemicals', axis = 1, inplace = True)
```

[6]:
```
df.info()
df.head()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 908 entries, 0 to 907
Data columns (total 7 columns):
 #   Column             Non-Null Count  Dtype
---  ------             --------------  -----
 0   CIC0               908 non-null    float64
 1   SM1_Dz(Z)          908 non-null    float64
 2   GATS1i             908 non-null    float64
 3   NdsCH              908 non-null    float64
 4   NdssC              908 non-null    float64
 5   MLOGP              908 non-null    float64
 6   LC50 [-LOG(mol/L)]  908 non-null    float64
dtypes: float64(7)
memory usage: 49.8 KB
```

[6]:
```
    CIC0  SM1_Dz(Z)  GATS1i  NdsCH  NdssC  MLOGP  LC50 [-LOG(mol/L)]
0  2.189      0.580   0.863    0.0    0.0  1.348               3.115
1  2.125      0.638   0.831    0.0    0.0  1.348               3.531
2  3.027      0.331   1.472    1.0    0.0  1.807               3.510
3  2.094      0.827   0.860    0.0    0.0  1.886               5.390
4  3.222      0.331   2.177    0.0    0.0  0.706               1.819
```

3 Exploratory Data Analysis

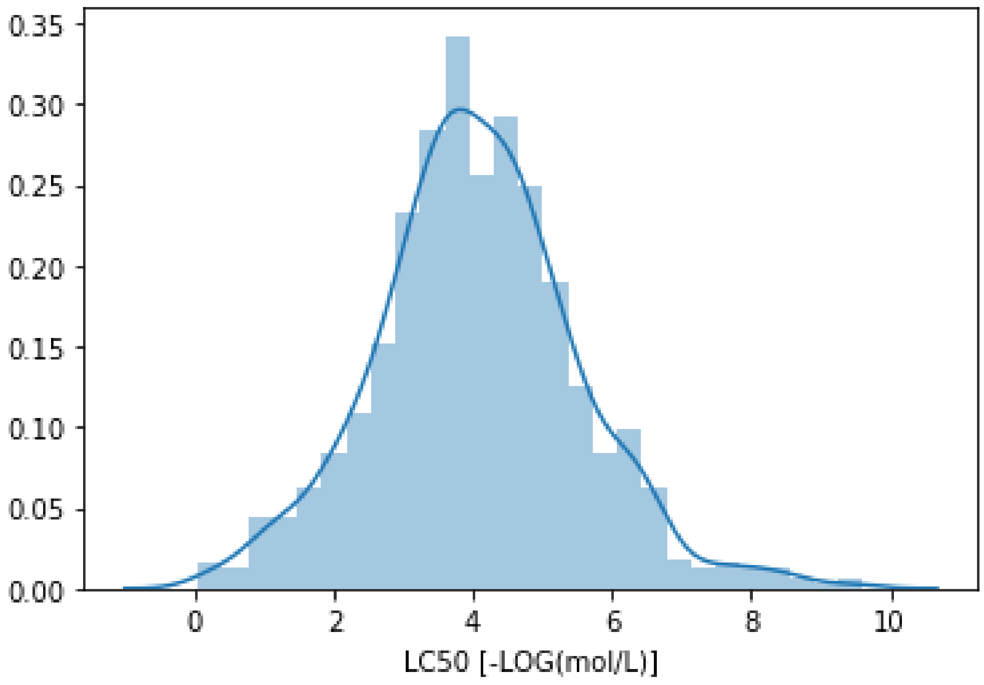
[7]:
```
df.describe()
```

[7]:
```
             CIC0   SM1_Dz(Z)      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 [-LOG(mol/L)]
count          908.000000
mean             4.064431
std              1.455698
min              0.053000
25%              3.151750
50%              3.987500
75%              4.907500
max              9.612000
```

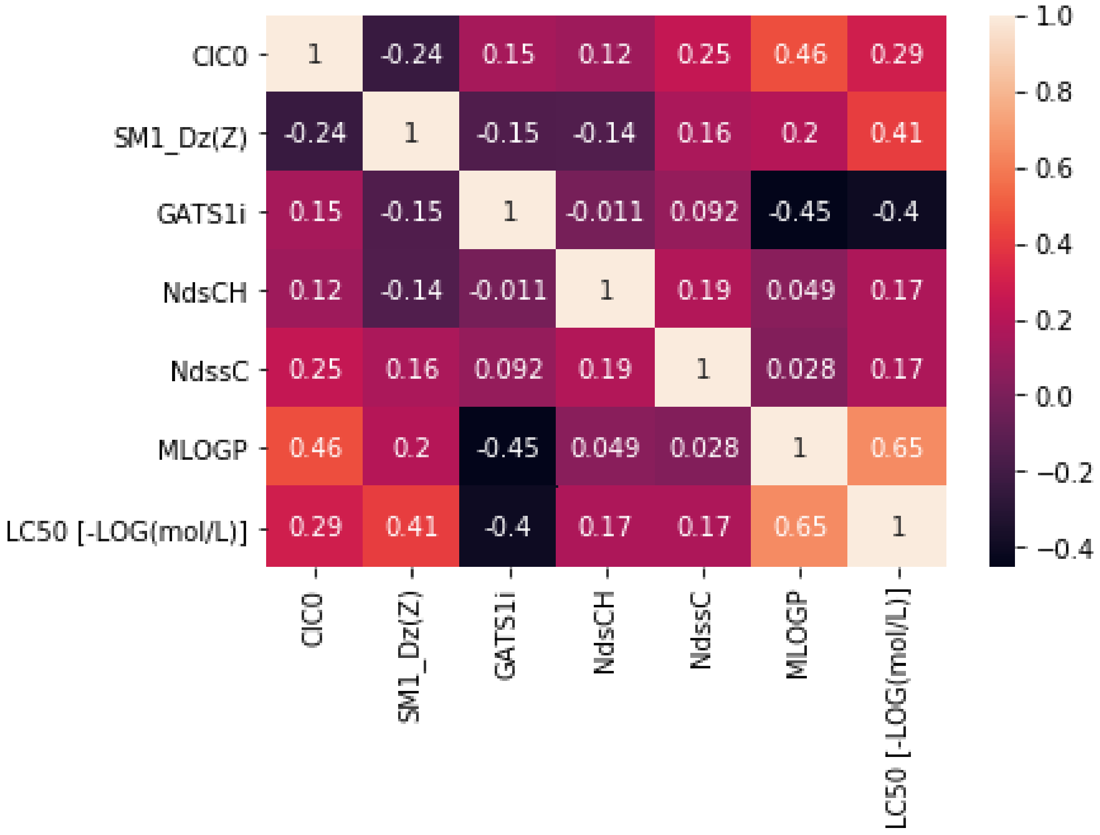
[8]:
```
sns.distplot(df['LC50 [-LOG(mol/L)]'], hist = True)
```

[8]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x1a1b34f410>
```



[9]:
```
sns.heatmap(df.corr(), annot=True)
```

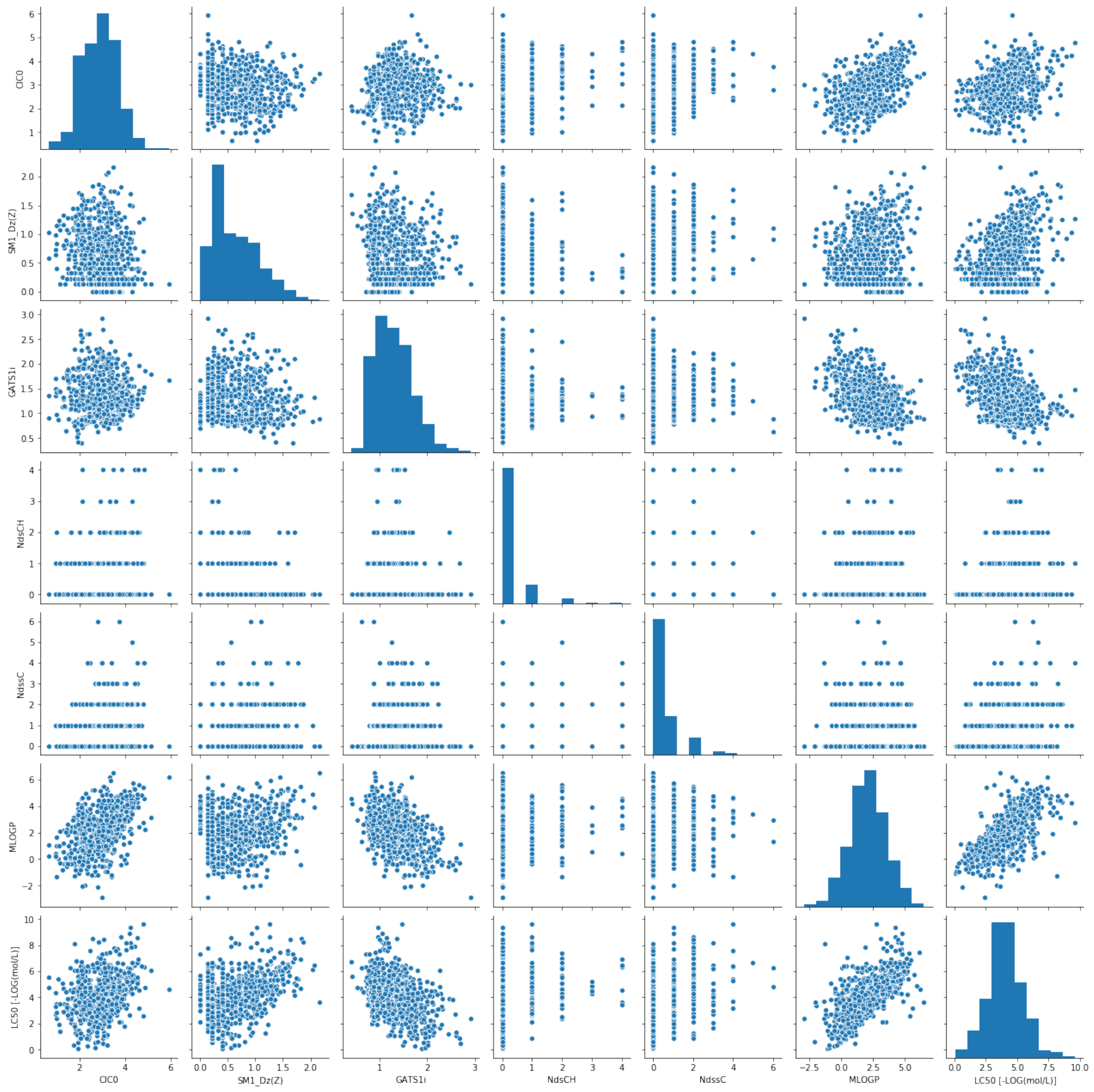
[9]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x1a1bbfd9d0>
```



	CIC0	SM1_Dz(Z)	GATS1i	NdsCH	NdssC	MLOGP	LC50 [-LOG(mol/L)]
CIC0	1	-0.24	0.15	0.12	0.25	0.46	0.29
SM1_Dz(Z)	-0.24	1	-0.15	-0.14	0.16	0.2	0.41
GATS1i	0.15	-0.15	1	-0.011	0.092	-0.45	-0.4
NdsCH	0.12	-0.14	-0.011	1	0.19	0.049	0.17
NdssC	0.25	0.16	0.092	0.19	1	0.028	0.17
MLOGP	0.46	0.2	-0.45	0.049	0.028	1	0.65
LC50 [-LOG(mol/L)]	0.29	0.41	-0.4	0.17	0.17	0.65	1

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

```
[10]: <seaborn.axisgrid.PairGrid at 0x1a1bd0fb90>
```



From the simple exploratory graphs above, it is shown that there is no clear linear relationship between the quantity of one particular molecular descriptor and the response variate LC_{50}, with the exception of MLOGP, which has a 0.65 correlation with LC_{50}.

In the following, Support Vector Machine Regression (SVR) in Scikit learn is used to build a prediction model, with the alternative of an Artificial Neural Network (ANN) from Tensorflow. Finally, the best performing model is adjusted with Principal Component Analysis (PCA) in order to reduce dimension and increase the speed of the algorithm.

4 Support Vector Regression (SVR)

First, split the data into a training set and a testing set.

```
[11]: from sklearn.model_selection import train_test_split
      X = df.drop('LC50 [-LOG(mol/L)]', axis = 1)
      y = df['LC50 [-LOG(mol/L)]']
      X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.30,␣
       ↪random_state=101)
```

Then, build and fit a SVR model to the training data. First use a linear kernel, with the alternative of the radial basis function kernel.

```
[12]: from sklearn.svm import SVR
      svr = SVR(kernel='linear')
      svr.fit(X_train, y_train)
```

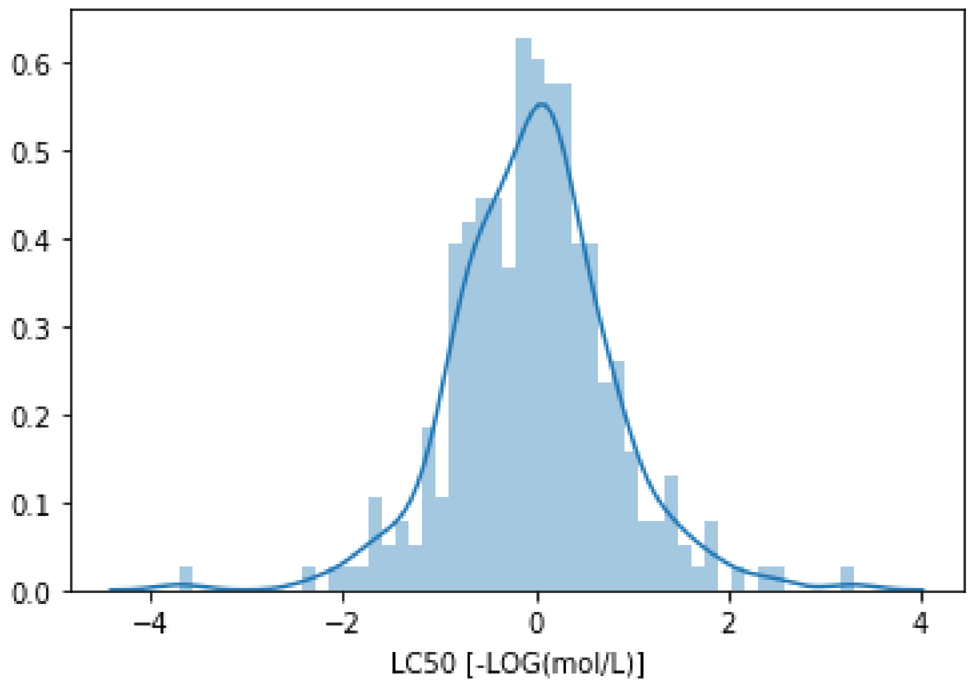
```
[12]: SVR(C=1.0, cache_size=200, coef0=0.0, degree=3, epsilon=0.1, gamma='scale',
          kernel='linear', max_iter=-1, shrinking=True, tol=0.001, verbose=False)
```

Now, use the SVR model to predict, and evaluate its performance.

```
[13]: from sklearn.metrics import mean_absolute_error, mean_squared_error
      predictions = svr.predict(X_test)
      print('MAE:', mean_absolute_error(y_test, predictions))
      print('MSE:', mean_squared_error(y_test, predictions))
      print('RMSE:', np.sqrt(mean_squared_error(y_test, predictions)))
      sns.distplot(y_test - predictions, bins = 50)
```

```
MAE: 0.61067591874093
MSE: 0.6789591017709762
RMSE: 0.8239897461564532
```

```
[13]: <matplotlib.axes._subplots.AxesSubplot at 0x1a1d87b510>
```

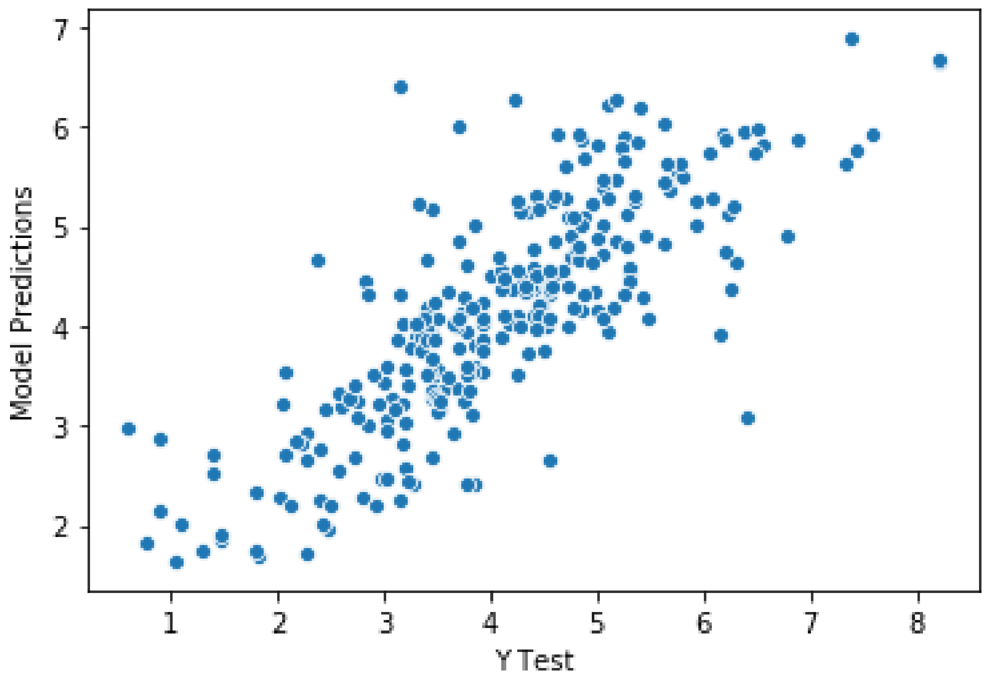


In the second attempt, apply a radial basis function kernel to the SVR model, and evaluate its prediction on the testing data.

```
[14]: svr = SVR(kernel='rbf')
      svr.fit(X_train, y_train)
      predictions = svr.predict(X_test)
```

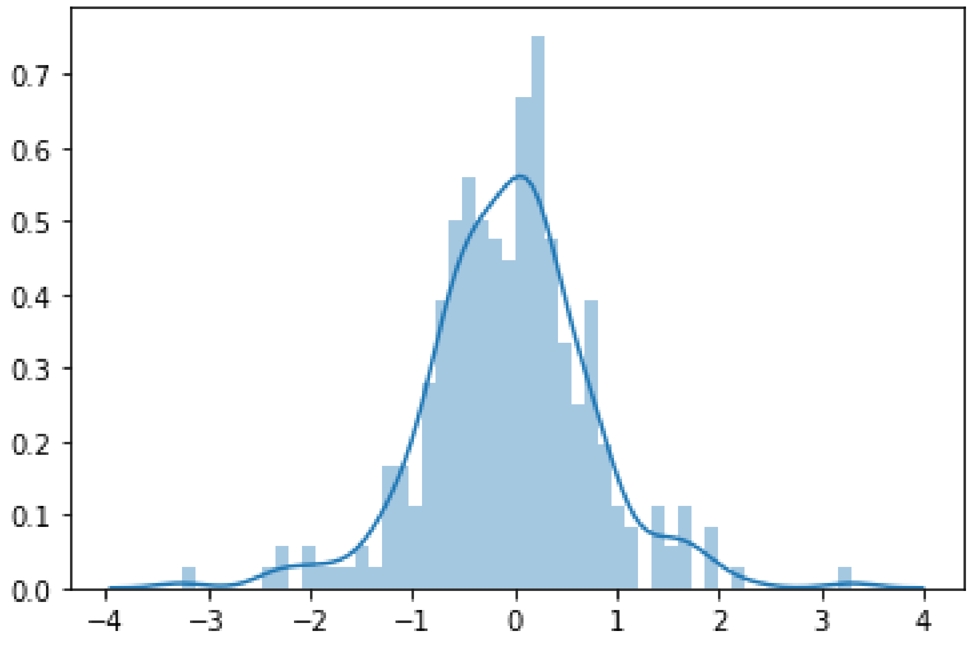
```
[19]: predDF = pd.DataFrame(y_test.values, columns=['Y Test'])
      prediction = pd.Series(predictions.reshape(273,))
      predDF = pd.concat([predDF, prediction], axis = 1)
      predDF.columns = ['Y Test','Model Predictions']
      sns.scatterplot(x='Y Test', y = 'Model Predictions', data = predDF)
```

```
[19]: <matplotlib.axes._subplots.AxesSubplot at 0x1a1e799cd0>
```



```
[20]: sns.distplot(predDF['Y Test'] - predDF['Model Predictions'], bins = 50)
```

```
[20]: <matplotlib.axes._subplots.AxesSubplot at 0x1a1e9d7c10>
```



```
[21]: print('MAE:', mean_absolute_error(y_test, predictions))
      print('MSE:', mean_squared_error(y_test, predictions))
      print('RMSE:', np.sqrt(mean_squared_error(y_test, predictions)))
```

```
MAE: 0.5970598256116714
MSE: 0.648675809701318
RMSE: 0.8054041281874076
```

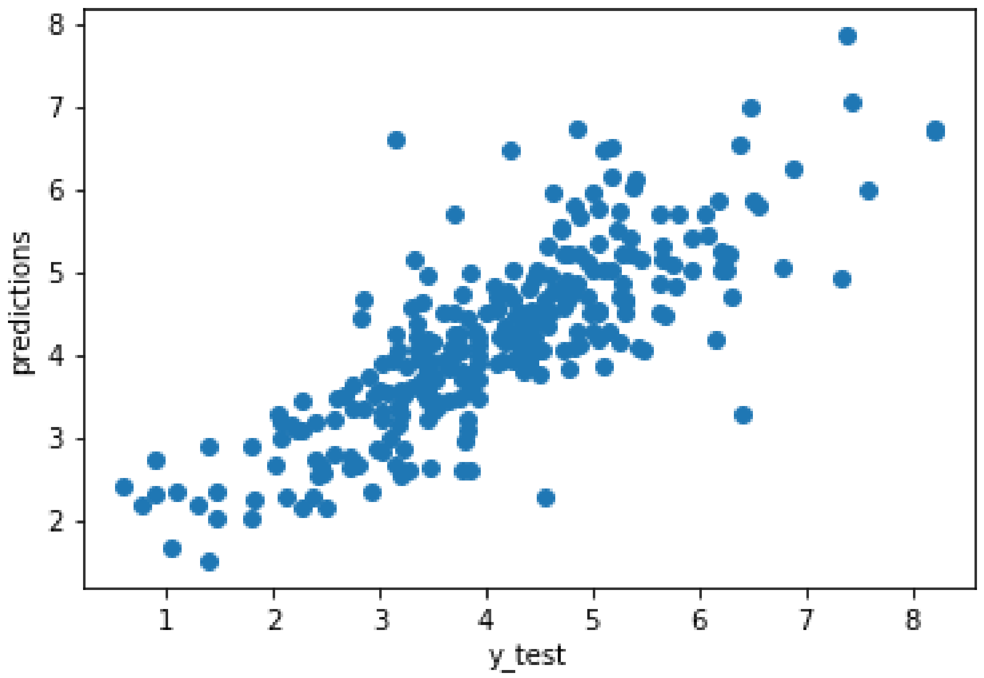
Note that this model is slightly better than the SVR model with linear kernel. Currently, this model is accepted as the best model. However, the RMSE of this model is not ideal (which has approximately 19.8% error). In the following, linear regression will be attempted.

5 Linear Regression

Attempt linear regression on the training data and evaluate its prediction performance on the testing data.

```
[22]: from sklearn.linear_model import LinearRegression
      lm = LinearRegression()
      lm.fit(X_train, y_train)
      predictions = lm.predict(X_test)
      plt.scatter(y_test, predictions)
      plt.xlabel('y_test')
      plt.ylabel('predictions')
```

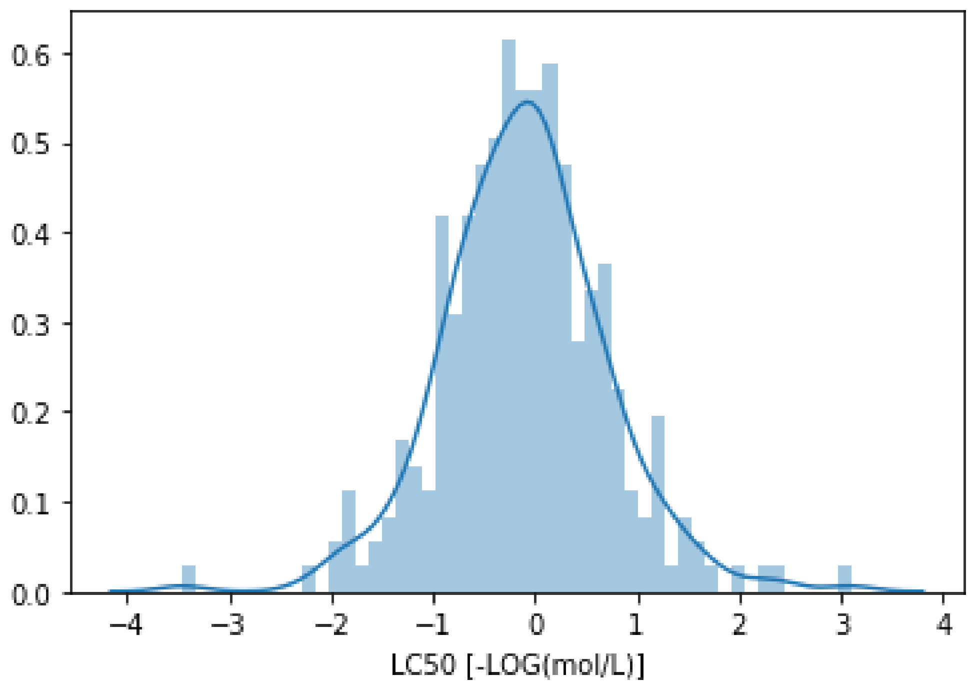
```
[22]: Text(0, 0.5, 'predictions')
```



[23]:
```
print('MAE:', mean_absolute_error(y_test, predictions))
print('MSE:', mean_squared_error(y_test, predictions))
print('RMSE:', np.sqrt(mean_squared_error(y_test, predictions)))
sns.distplot(y_test - predictions, bins = 50)
```

```
MAE: 0.6129943145241861
MSE: 0.660646467821266
RMSE: 0.8128016165223012
```

[23]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x1a1ebf6f90>
```



Note that this linear regression model is slightly better than SVR with linear kernel, but is slightly worse than SVR with radial basis function kernel, which is currently the best performing model. In the following, an ANN from Tensorflow is attempted to improve model performance.

6 Artificial Neural Network

First, data needs to be scaled to standardize the variance.

```
[24]: from sklearn.preprocessing import MinMaxScaler
      scaler = MinMaxScaler()
      X_train = scaler.fit_transform(X_train)
      X_test = scaler.transform(X_test)
      X_train.shape, X_test.shape
```

```
[24]: ((635, 6), (273, 6))
```

Next, build various ANN models in the following order: - Rmsprop (gradient descent) optimizer, with Early Stopping mechanism. - Rmsprop (gradient descent) optimizer, with Early Stopping mechanism and Dropout layers. - Adam optimizer, with Early Stopping mechanism.

First, attempt the Rmsprop optimizer with Early Stopping but without Dropout layers.

```
[26]: from tensorflow.keras.models import Sequential
      from tensorflow.keras.layers import Dense, Activation, Dropout
      from tensorflow.keras.optimizers import Adam
      from tensorflow.keras.callbacks import EarlyStopping

      model = Sequential()

      model.add(Dense(6, activation = 'relu'))
      model.add(Dense(6, activation = 'relu'))
      model.add(Dense(6, activation = 'relu'))
      model.add(Dense(6, activation = 'relu'))

      model.add(Dense(1))

      model.compile(optimizer = 'rmsprop', loss = 'mse')
```

```
[27]: earlyStop = EarlyStopping(monitor='val_loss', mode = 'min', verbose = 1,␣
        ↪patience=20)
      model.fit(x = X_train,
                y = y_train.values,
                validation_data=(X_test, y_test.values),
                batch_size = 64,
                epochs=500,
                verbose = 1,
                callbacks = [earlyStop])
```

```
Train on 635 samples, validate on 273 samples
Epoch 1/500
635/635 [==============================] - 1s 983us/sample - loss: 18.6790 -
val_loss: 18.2099
Epoch 2/500
```

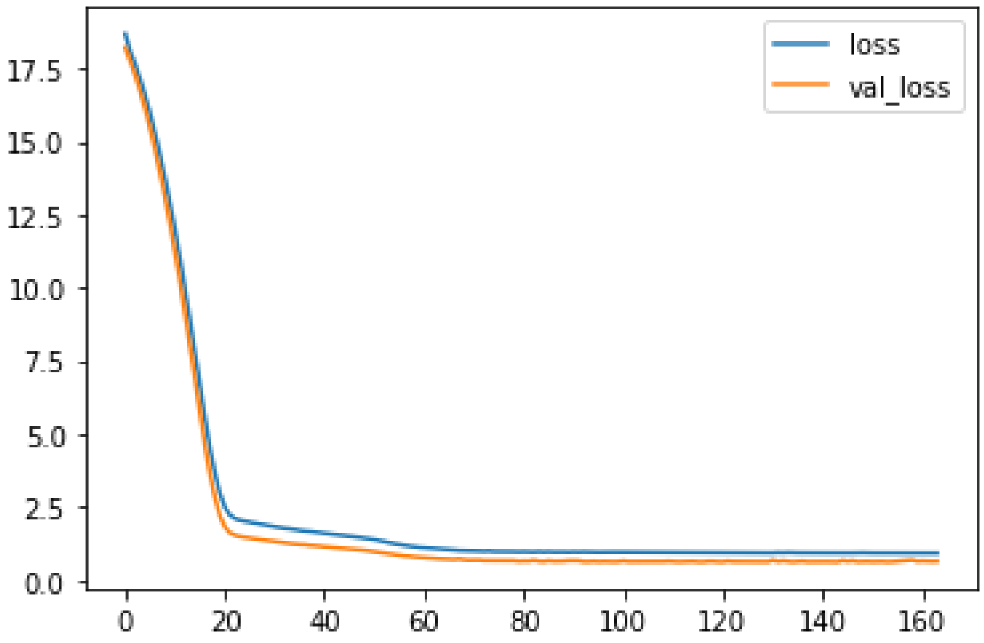
```
635/635 [==============================] - 0s 29us/sample - loss: 18.0952 -
val_loss: 17.7551
Epoch 3/500
635/635 [==============================] - 0s 27us/sample - loss: 17.6418 -
val_loss: 17.2726
Epoch 4/500
635/635 [==============================] - 0s 29us/sample - loss: 17.1434 -
val_loss: 16.7239
Epoch 5/500
635/635 [==============================] - 0s 29us/sample - loss: 16.5873 -
val_loss: 16.1203
Epoch 158/500
635/635 [==============================] - 0s 27us/sample - loss: 0.9235 -
val_loss: 0.6866
Epoch 159/500
635/635 [==============================] - 0s 26us/sample - loss: 0.9277 -
val_loss: 0.6947
Epoch 160/500
635/635 [==============================] - 0s 26us/sample - loss: 0.9328 -
val_loss: 0.6448
Epoch 161/500
635/635 [==============================] - 0s 29us/sample - loss: 0.9237 -
val_loss: 0.6554
Epoch 162/500
635/635 [==============================] - 0s 29us/sample - loss: 0.9238 -
val_loss: 0.6469
Epoch 163/500
635/635 [==============================] - 0s 30us/sample - loss: 0.9219 -
val_loss: 0.6410
Epoch 164/500
635/635 [==============================] - 0s 29us/sample - loss: 0.9240 -
val_loss: 0.6475
Epoch 00164: early stopping
```

```
[27]: <tensorflow.python.keras.callbacks.History at 0x1a3b1ddfd0>
```

Monitor the training process and evaluate the model performance.

```
[28]: modelLoss = pd.DataFrame(model.history.history)
      modelLoss.plot()
```

```
[28]: <matplotlib.axes._subplots.AxesSubplot at 0x1a3b96ff10>
```



[29]:
```
trainScore = model.evaluate(X_train, y_train, verbose = 0)
testScore = model.evaluate(X_test, y_test, verbose = 0)
trainScore, testScore
```

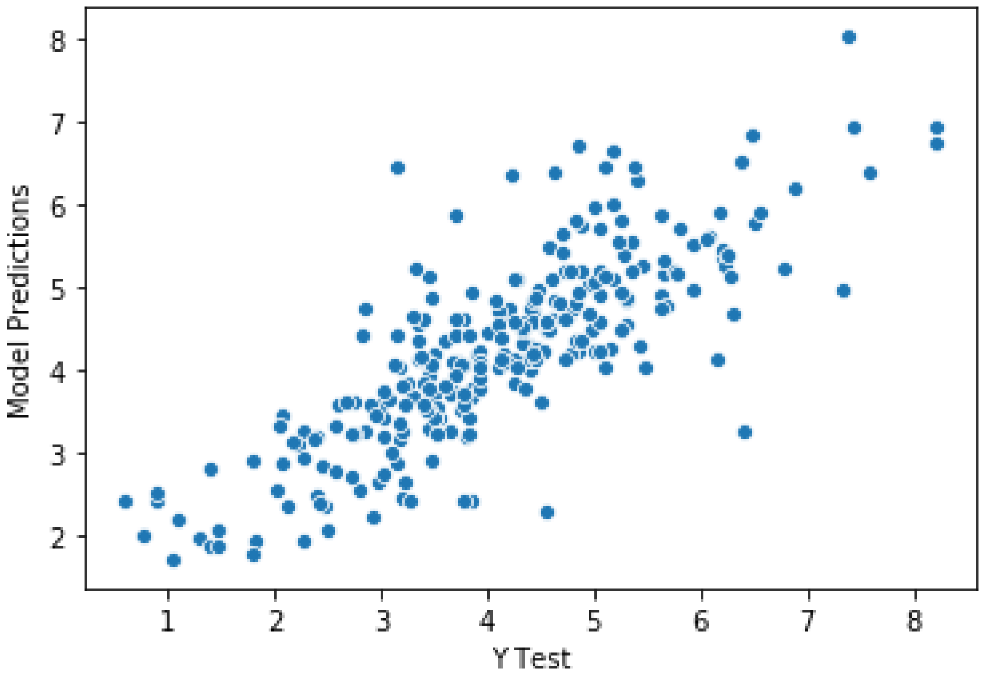
[29]:
```
(0.9186066175070334, 0.6474737444203416)
```

Notice that the scores have approximately 0.3 difference, which is considerable for a response variate with a mean of approximately 4.

Further evaluate the model prediction performance.

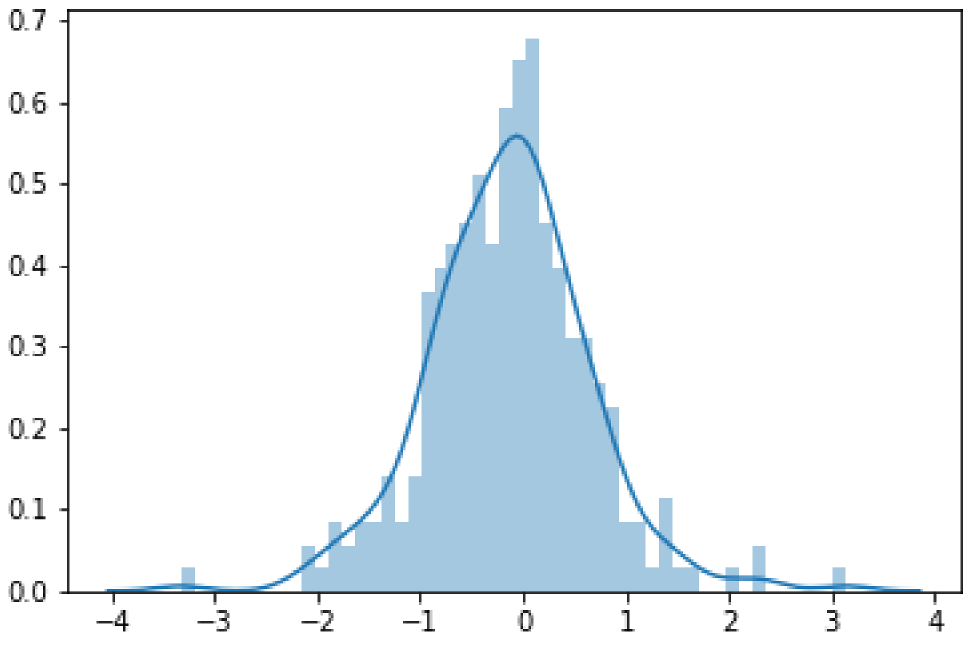
[32]:
```
prediction = model.predict(X_test)
predDF = pd.DataFrame(y_test.values, columns=['Y Test'])
prediction = pd.Series(prediction.reshape(273,))
predDF = pd.concat([predDF, prediction], axis = 1)
predDF.columns = ['Y Test','Model Predictions']
sns.scatterplot(x='Y Test', y = 'Model Predictions', data = predDF)
```

[32]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x1a3bb018d0>
```



```
[33]: sns.distplot(predDF['Y Test'] - predDF['Model Predictions'], bins = 50)
```

```
[33]: <matplotlib.axes._subplots.AxesSubplot at 0x1a3bae27d0>
```



```
[34]: # MAE, MSE, RMSE
print(mean_absolute_error(predDF['Y Test'],predDF['Model Predictions']))
print(mean_squared_error(predDF['Y Test'],predDF['Model Predictions']))
print(np.sqrt(mean_squared_error(predDF['Y Test'],predDF['Model Predictions'])))
```

```
0.6022623935950983
0.6474737390857223
0.8046575290679397
```

Note that the errors are greater than that of the current best model. In the following, Dropout layers are added in attempt to improve the model.

```
[43]: model = Sequential()

model.add(Dense(6, activation = 'relu'))
model.add(Dropout(0.2))

model.add(Dense(6, activation = 'relu'))
model.add(Dropout(0.2))

model.add(Dense(6, activation = 'relu'))
model.add(Dropout(0.2))

model.add(Dense(6, activation = 'relu'))
model.add(Dropout(0.2))

model.add(Dense(1))

model.compile(optimizer = 'rmsprop', loss = 'mse')
```

```
[44]: earlyStop = EarlyStopping(monitor='val_loss', mode = 'min', verbose = 1,␣
 ↪patience=20)
model.fit(x = X_train,
          y = y_train.values,
          validation_data=(X_test, y_test.values),
          batch_size = 64,
          epochs=500,
          verbose = 1,
          callbacks = [earlyStop])
```

```
Train on 635 samples, validate on 273 samples
Epoch 1/500
635/635 [==============================] - 0s 741us/sample - loss: 18.4496 -
val_loss: 18.1627
Epoch 2/500
635/635 [==============================] - 0s 29us/sample - loss: 18.0779 -
val_loss: 17.8081
Epoch 3/500
```

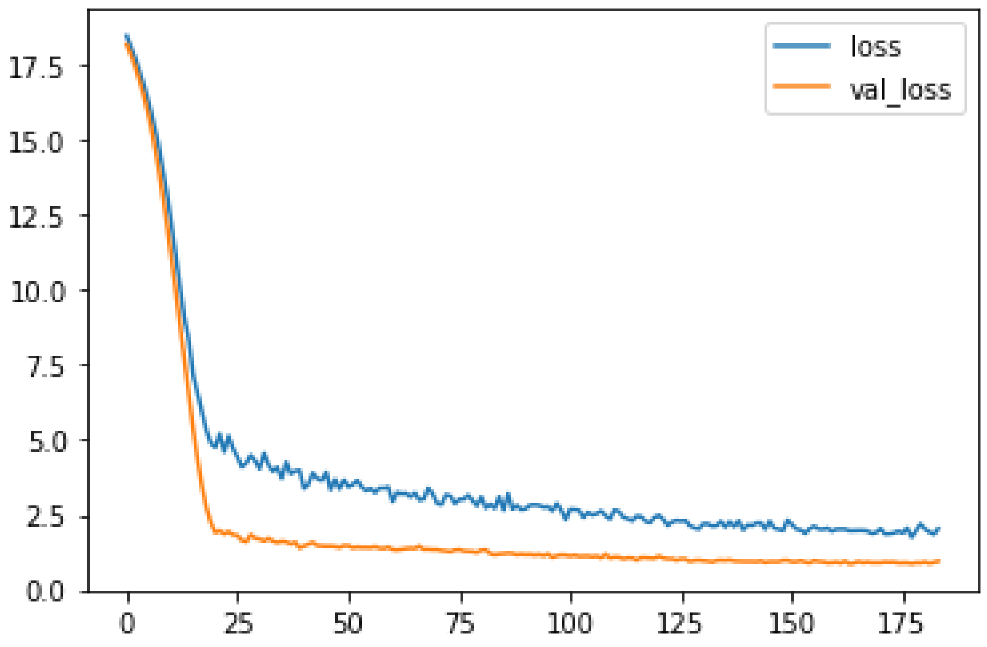
```
635/635 [==============================] - 0s 30us/sample - loss: 17.6920 -
val_loss: 17.3993
Epoch 4/500
635/635 [==============================] - 0s 31us/sample - loss: 17.2452 -
val_loss: 16.9203
Epoch 5/500
635/635 [==============================] - 0s 32us/sample - loss: 16.7683 -
val_loss: 16.3862
Epoch 179/500
635/635 [==============================] - 0s 28us/sample - loss: 1.9720 -
val_loss: 0.9074
Epoch 180/500
635/635 [==============================] - 0s 28us/sample - loss: 2.1955 -
val_loss: 0.8992
Epoch 181/500
635/635 [==============================] - 0s 28us/sample - loss: 2.0354 -
val_loss: 0.9353
Epoch 182/500
635/635 [==============================] - 0s 27us/sample - loss: 1.9288 -
val_loss: 0.8839
Epoch 183/500
635/635 [==============================] - 0s 27us/sample - loss: 1.8540 -
val_loss: 0.9161
Epoch 184/500
635/635 [==============================] - 0s 27us/sample - loss: 2.0407 -
val_loss: 0.9598
Epoch 00184: early stopping
```

```
[44]: <tensorflow.python.keras.callbacks.History at 0x1a3ca66cd0>
```

Monitor the training process and evaluate the model performance.

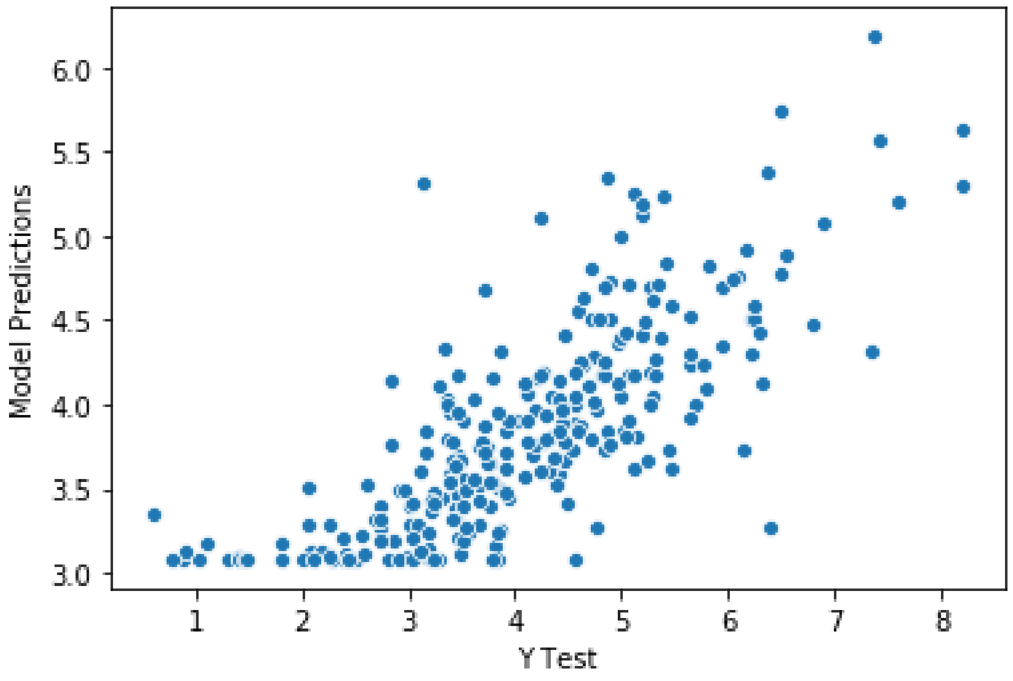
```
[45]: modelLoss = pd.DataFrame(model.history.history)
      modelLoss.plot()
```

```
[45]: <matplotlib.axes._subplots.AxesSubplot at 0x1a3cc6b590>
```



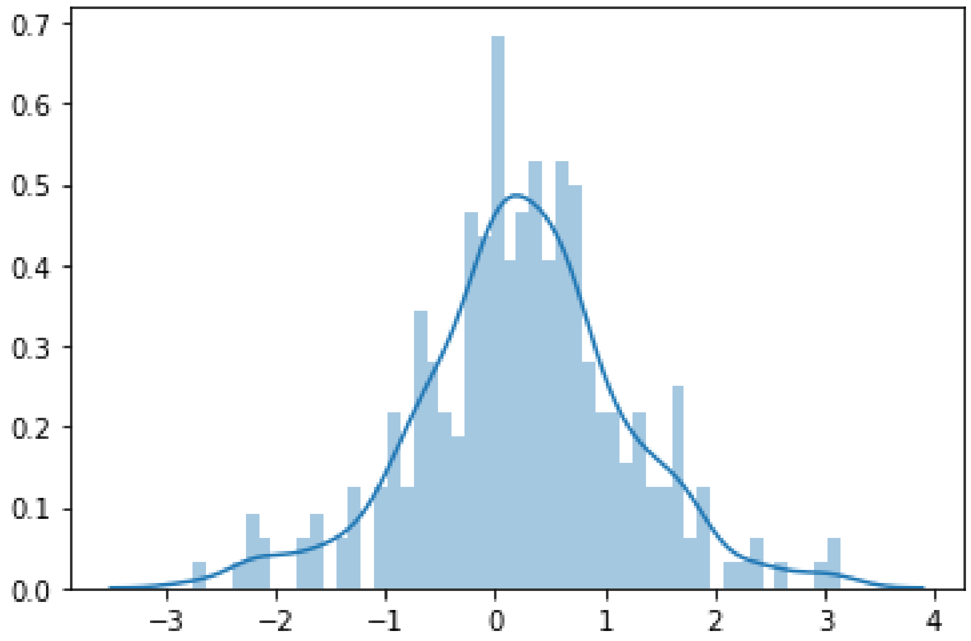
```
[46]: prediction = model.predict(X_test)
      predDF = pd.DataFrame(y_test.values, columns=['Y Test'])
      prediction = pd.Series(prediction.reshape(273,))
      predDF = pd.concat([predDF, prediction], axis = 1)
      predDF.columns = ['Y Test','Model Predictions']
      sns.scatterplot(x='Y Test', y = 'Model Predictions', data = predDF)
```

```
[46]: <matplotlib.axes._subplots.AxesSubplot at 0x1a3b924c90>
```



```
[47]: sns.distplot(predDF['Y Test'] - predDF['Model Predictions'], bins = 50)
```

```
[47]: <matplotlib.axes._subplots.AxesSubplot at 0x1a3c6d8690>
```



```
[48]: # MAE, MSE, RMSE
      print(mean_absolute_error(predDF['Y Test'],predDF['Model Predictions']))
      print(mean_squared_error(predDF['Y Test'],predDF['Model Predictions']))
      print(np.sqrt(mean_squared_error(predDF['Y Test'],predDF['Model Predictions'])))
```

```
0.7435311851990529
0.9598180468919105
0.9797030401565112
```

Note that the error for the model with Dropout layers are much higher than that of the previous models. Thus, an ANN with adam optimizer but without the dropout layers is attempted.

```
[49]: model = Sequential()

      model.add(Dense(6, activation = 'relu'))

      model.add(Dense(6, activation = 'relu'))

      model.add(Dense(6, activation = 'relu'))

      model.add(Dense(6, activation = 'relu'))

      model.add(Dense(1))

      model.compile(optimizer = 'adam', loss = 'mse')
```

```
[50]: earlyStop = EarlyStopping(monitor='val_loss', mode = 'min', verbose = 1,␣
       ↪patience=20)
      model.fit(x = X_train,
                y = y_train.values,
                validation_data=(X_test, y_test.values),
                batch_size = 64,
                epochs=500,
                verbose = 1,
                callbacks = [earlyStop])
```

```
Train on 635 samples, validate on 273 samples
Epoch 1/500
635/635 [==============================] - 0s 450us/sample - loss: 18.6561 -
val_loss: 18.4357
Epoch 2/500
635/635 [==============================] - 0s 24us/sample - loss: 18.4161 -
val_loss: 18.1977
Epoch 3/500
635/635 [==============================] - 0s 26us/sample - loss: 18.1680 -
val_loss: 17.9268
Epoch 4/500
635/635 [==============================] - 0s 29us/sample - loss: 17.8789 -
```

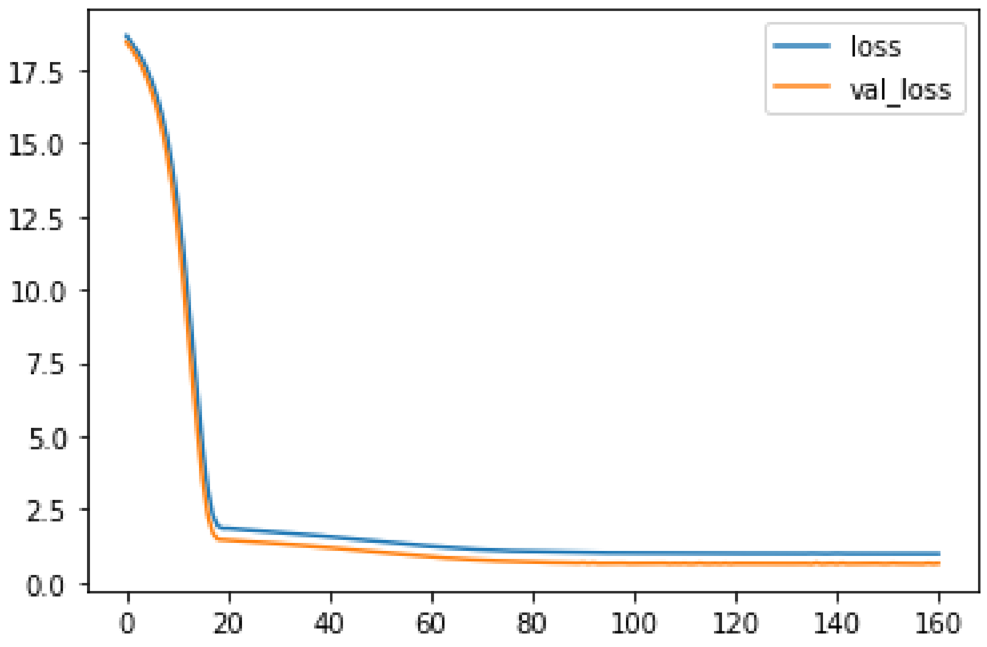
```
val_loss: 17.6013
Epoch 5/500
635/635 [==============================] - 0s 29us/sample - loss: 17.5265 -
val_loss: 17.2078
Epoch 156/500
635/635 [==============================] - 0s 32us/sample - loss: 0.9791 -
val_loss: 0.6484
Epoch 157/500
635/635 [==============================] - 0s 31us/sample - loss: 0.9798 -
val_loss: 0.6471
Epoch 158/500
635/635 [==============================] - 0s 29us/sample - loss: 0.9798 -
val_loss: 0.6353
Epoch 159/500
635/635 [==============================] - 0s 28us/sample - loss: 0.9797 -
val_loss: 0.6574
Epoch 160/500
635/635 [==============================] - 0s 27us/sample - loss: 0.9831 -
val_loss: 0.6361
Epoch 161/500
635/635 [==============================] - 0s 26us/sample - loss: 0.9804 -
val_loss: 0.6514
Epoch 00161: early stopping
```

```
[50]: <tensorflow.python.keras.callbacks.History at 0x1a3b19d450>
```

Monitor the training process and evaluate the model performance.

```
[52]: modelLoss = pd.DataFrame(model.history.history)
      modelLoss.plot()
```

```
[52]: <matplotlib.axes._subplots.AxesSubplot at 0x1a3ca669d0>
```

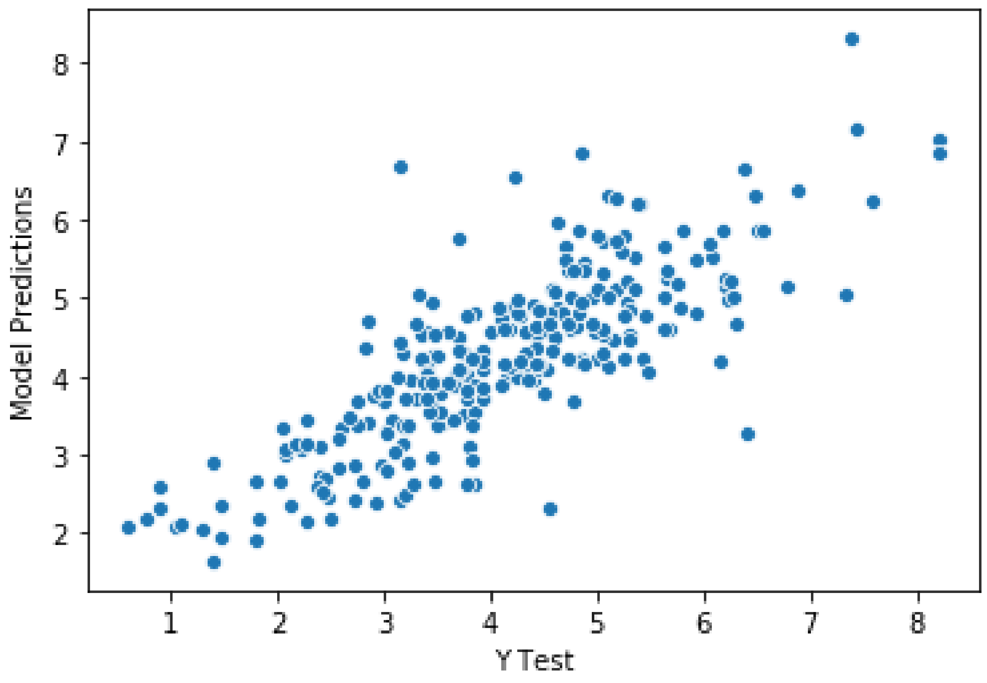


[53]:
```
trainScore = model.evaluate(X_train, y_train, verbose = 0)
testScore = model.evaluate(X_test, y_test, verbose = 0)
trainScore, testScore
```

[53]:
```
(0.9760275619236503, 0.6514134778207912)
```

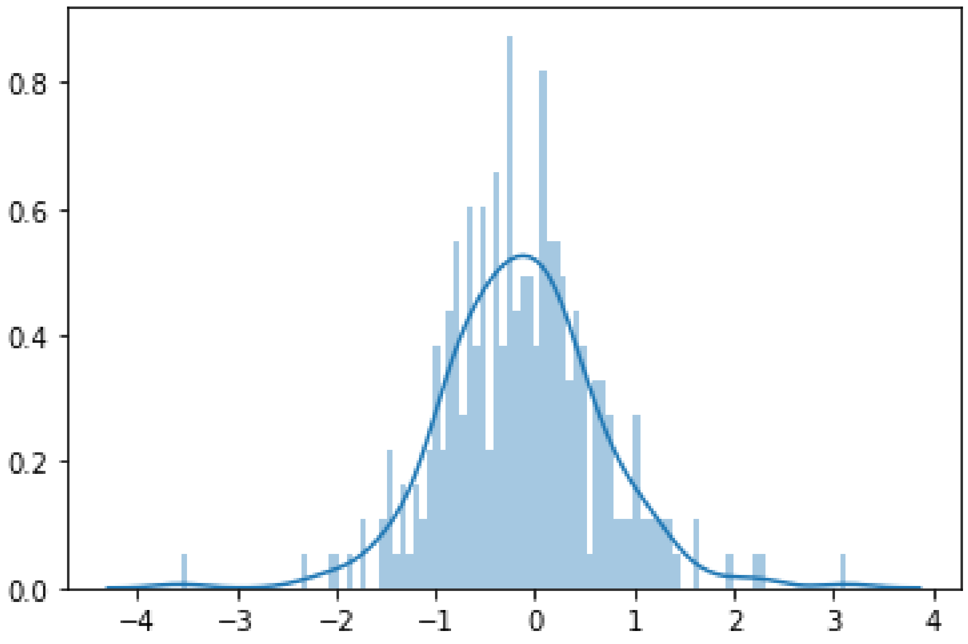
[54]:
```
prediction = model.predict(X_test)
predDF = pd.DataFrame(y_test.values, columns=['Y Test'])
prediction = pd.Series(prediction.reshape(273,))
predDF = pd.concat([predDF, prediction], axis = 1)
predDF.columns = ['Y Test','Model Predictions']
sns.scatterplot(x='Y Test', y = 'Model Predictions', data = predDF)
```

[54]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x1a3d166c10>
```



```
[55]: sns.distplot(predDF['Y Test'] - predDF['Model Predictions'], bins = 100)
```

```
[55]: <matplotlib.axes._subplots.AxesSubplot at 0x1a3d13c850>
```



```
[56]: # MAE, MSE, RMSE
print(mean_absolute_error(predDF['Y Test'],predDF['Model Predictions']))
print(mean_squared_error(predDF['Y Test'],predDF['Model Predictions']))
print(np.sqrt(mean_squared_error(predDF['Y Test'],predDF['Model Predictions'])))
```

```
0.6151891960521321
0.6514135038024089
0.8071019166142581
```

Note that this is still worse than the current best model, the SVR model with a radial basis function kernel, which would now be accepted as the best model for this dataset.

7 Support Vector Regression Revisited

The support vector regression model is experimented to have the best performance. In the following, Principal Component Analysis (PCA) will be attempted to improve the model. First, observe the current best model:

```
[61]: X = df.drop('LC50 [-LOG(mol/L)]', axis = 1)
y = df['LC50 [-LOG(mol/L)]']
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.30,␣
 ↪random_state=101)
```

```
[62]: svr = SVR(kernel='rbf')
svr.fit(X_train, y_train)
predictions = svr.predict(X_test)
```

```
[63]: print('MAE:', mean_absolute_error(y_test, predictions))
print('MSE:', mean_squared_error(y_test, predictions))
print('RMSE:', np.sqrt(mean_squared_error(y_test, predictions)))
```

```
MAE: 0.5970598256116714
MSE: 0.648675809701318
RMSE: 0.8054041281874076
```

Attempting to use PCA to reduce dimension and speed up the algorithm gives:

```
[65]: from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
scaler.fit(df.drop('LC50 [-LOG(mol/L)]', axis =1))
scaled_data = scaler.transform(df.drop('LC50 [-LOG(mol/L)]', axis =1))
```

Split the data, then train the PCA model on the training data only. Finally, apply the PCA model on both the training data and the testing data to reduce their dimensions.

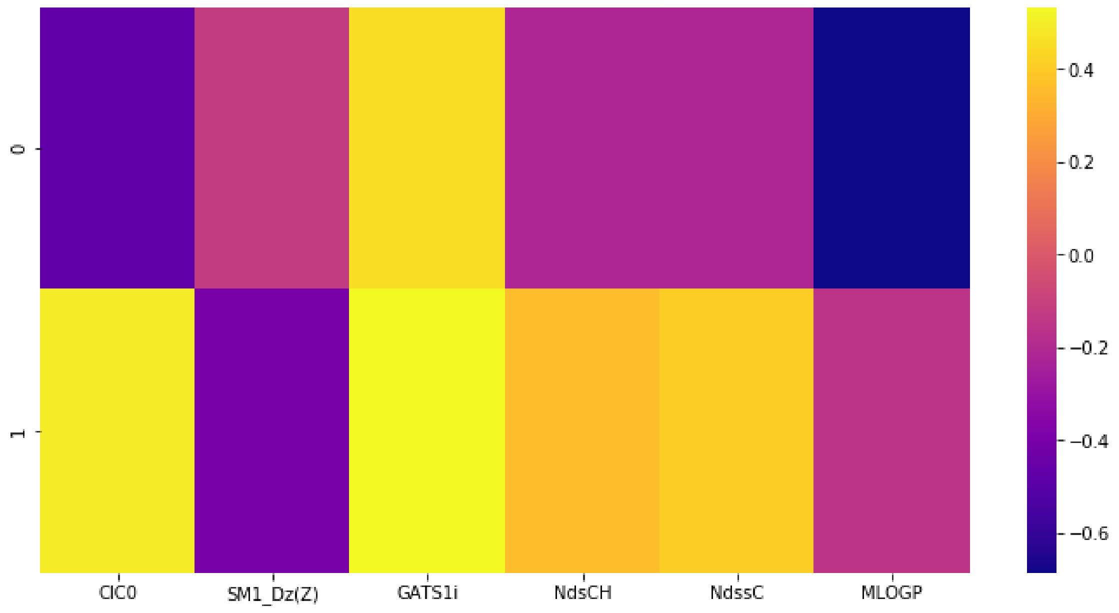
```
[67]: X = df.drop('LC50 [-LOG(mol/L)]', axis = 1)
      y = df['LC50 [-LOG(mol/L)]']
      X_train, X_test, y_train, y_test = train_test_split(scaled_data,
                                                          y,
                                                          test_size=0.30,
                                                          random_state=101)
```

```
[68]: from sklearn.decomposition import PCA
      pca = PCA(n_components=2)
      pca.fit(X_train)
      xTrainPCA = pca.transform(X_train)
      xTestPCA = pca.transform(X_test)
```

The relatinoship between the PCA indicator and the six molecular descriptors are shown below:

```
[69]: df_comp = pd.DataFrame(pca.components_, columns=df.drop('LC50 [-LOG(mol/L)]',␣
       ↪axis =1).columns)
      plt.figure(figsize = (12, 6))
      sns.heatmap(df_comp, cmap='plasma')
```

```
[69]: <matplotlib.axes._subplots.AxesSubplot at 0x1a3d345f50>
```



In this way, the PCA indicators, instead of the six molecular descriptors, can be used to train the SVR model with radial basis function kernel.

[70]:
```
svrPCA = SVR(kernel='rbf')
svrPCA.fit(xTrainPCA, y_train)
predictions = svrPCA.predict(xTestPCA)
```

[71]:
```
print('MAE:', mean_absolute_error(y_test, predictions))
print('MSE:', mean_squared_error(y_test, predictions))
print('RMSE:', np.sqrt(mean_squared_error(y_test, predictions)))
```

```
MAE: 0.6387268135017251
MSE: 0.7432998993485277
RMSE: 0.8621484207191519
```

Observe that the PCA indicators are not good at predicting the response variate comparing to the original six molecular descriptors. This confirms that the SVR model with radial basis function kernel, using the molecular descriptors provided, performs the best.

In the following, this model is used to predict a single entry.

[75]:
```
entry = df['LC50 [-LOG(mol/L)]'].iloc[904]
pred = svr.predict([df.drop('LC50 [-LOG(mol/L)]', axis = 1).iloc[904].values.
 ↪reshape(6, )])
print("The model-predicted toxin level is: ",  pred[0])
print("The actual toxin level is: " , entry)
```

```
The model-predicted toxin level is:  4.918001405697297
The actual toxin level is:  4.818
```

8 Conclusion

The research conducted by M. Cassotti et al. provides six molecular descriptors that have a certain degree of correlations with the response variate LC_{50}. In the experimental process described above, it is shown after evaluating the performance of various models including SVR with linear kernel and RBF kernel, linear regression, as well as ANN with Rmsprop and Adam optimizers, that the SVR model with radial basis function kernel, using the molecular descriptors provided, performs the best. The model has a Root Mean Squared Error (RMSE) of 0.805, which is an approximately 19.8% error of the response variate LC_{50}. While this error is considerable to some extent, it is shown with sufficient evidence that the six molecular descriptors are correlated to LC_{50} and can be used to predict the toxin level in fathead minnow.

[78]:
```
from joblib import dump
dump(svr, 'FatheadMinnowModel.h5')
```

[78]:
```
['FatheadMinnowModel.h5']
```