ECE 566 - Machine and Deep Learning

Homework #2 - Error Analysis
Due Date: 09/15/2022

Anas Puthawala
A20416308

Introduction

The purpose of this homework assignment is to familiarize the student with how to generate proper data for training, testing, etc. with noise added in along with evaluating different types of error. More specifically, the Generalization Error, Modeling Error, and Training Error were touched upon and the implementation of the mathematical formula for each of the error types were conducted in Python. A plot was generated with all three errors overlayed on top of one another with neighborhoods in the range of 1-35 reveals that the training error is not ideal for evaluating a model's performance. In most cases, the modeling error is not possible to evaluate since we'd have to know the actual model f(x), which leads to the generalization error being the best evaluator of a given model's performance.
Further, a KNN model was trained on the generated data with differing neighborhood sizes and the models were plotted with the testing data, and the actual model to visualize the performance of different neighborhood sizes as predictors the testing dataset.
It was found through both the plotting of the errors and the plotting of various K's as neighborhood sizes, that K=5 was the most optimal K to fit the curve.

Body

The first step consisted of generating the training data and testing data. The data generation process was straight-forward and not much needs to be explained. The next step consisted of evaluating different neighborhood sizes. The process for this was to simply train a KNN regressor on the given N_sizes_plot which were 1, 2, 5, 35 and visualize each of them 1. together and 2. individually overlayed with f(x), y_test, and the KNN regressor.

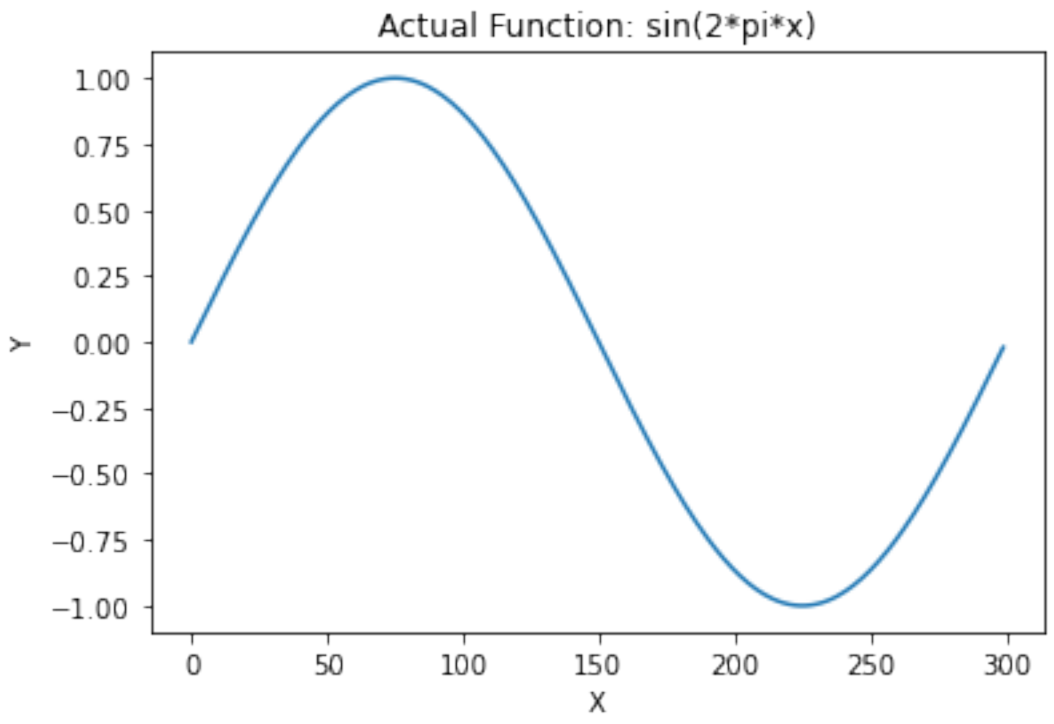

Figure 1. Actual Model
Figure 1 shows the actual model that we're going to be using to gauge predictions on. The better the fit towards the sin curve, the better the regressor. In reality, we wouldn't know this

model and so we estimated this model and our f_x_test and f_x_train (with some added noise) looks as such:

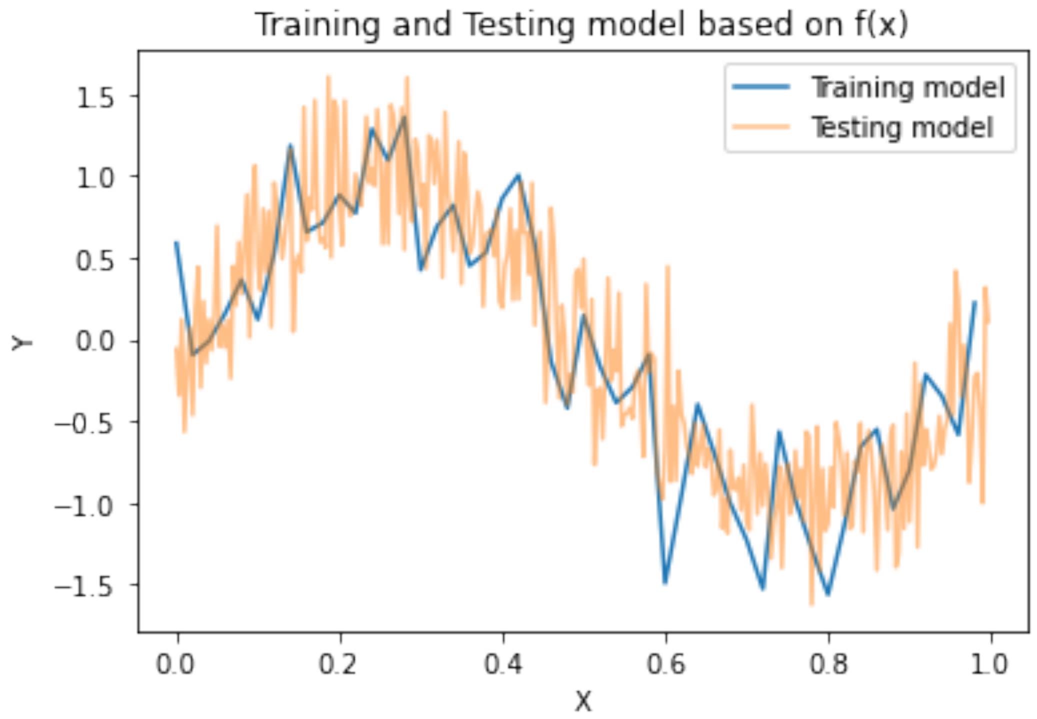

Figure 2. Training and testing model generated manually

The figure above shows our training and testing model which is generated manually. These are the models that we'd utilize to A. train the KNN regressor and B. evaluate performance (at-least for the training and generalization errors)

So next step is to go ahead and train the actual KNNRegressor with varying neighborhood sizes.

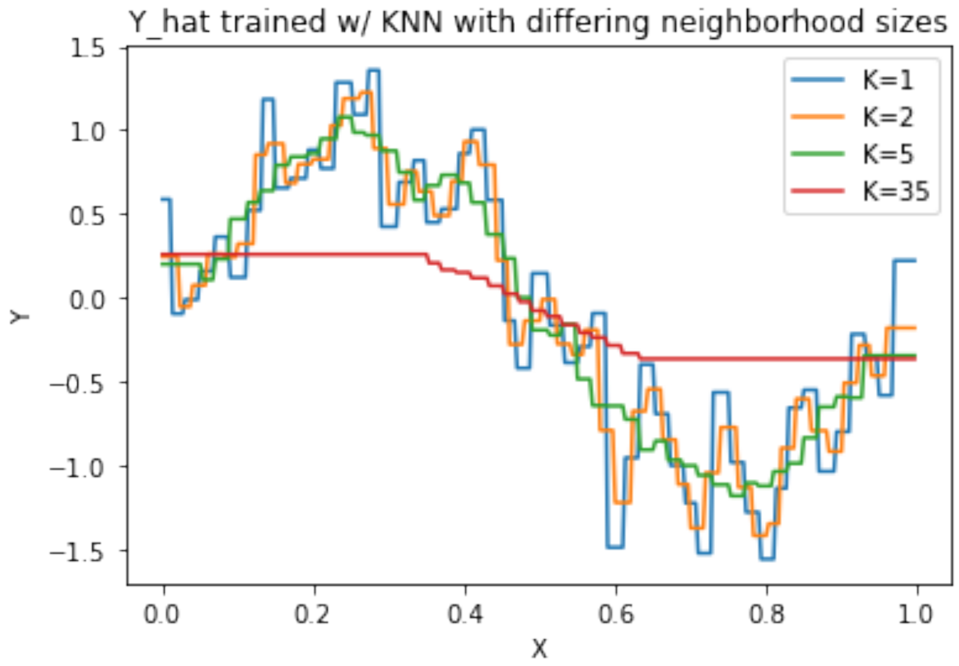

Figure 3. KNN regressor trained for different neighborhood sizes

Figure 3 shows the KNN regressor model trained for different neighborhood sizes. We can see right off the bat that K=5 is actually the optimal model to fit the data (it's the green line in the plot above). Next we can go ahead and plot the KNN models regressed for k = 1,2,5, and 35 along with the actual function f(x)   and y_test.

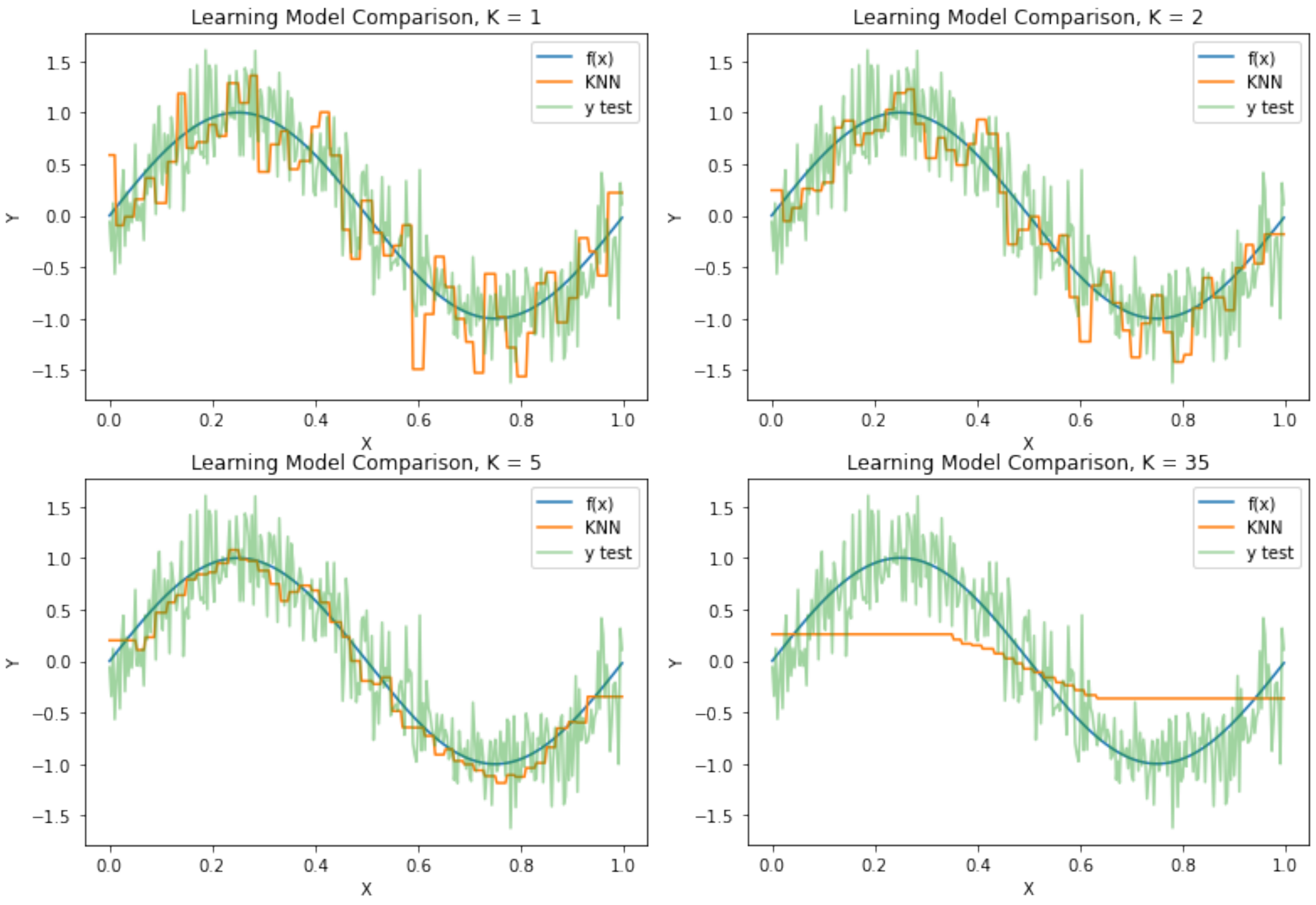

Figure 4. Learning model comparisons for differing neighborhood sizes

Starting with the top-left, you can see that with K=1 the model is very complex and has high variance as it tries to fit the training data too well (you can see the training data from figure 2, in the blue). For K=2 again the model has somewhat high variance and low bias, it's unable to learn more important characteristics and just overfits the data similar to K=1. When we get to K=5 we start to see a shift from the variance towards bias. It learns the overall curve that it needs to, without overfitting on the 'noise' that can be seen from the training data. K=5 is the better model out of this group as it has the perfect balance between bias and variance. When we look at K=35 we see very low variance, but also extremely high bias. This model is no good and fails to learn the properties of the actual curve itself.

Next, we'll confirm that K=5 is indeed the best model for this case by analyzing the plots of the generalization error, training error, and the modeling error.

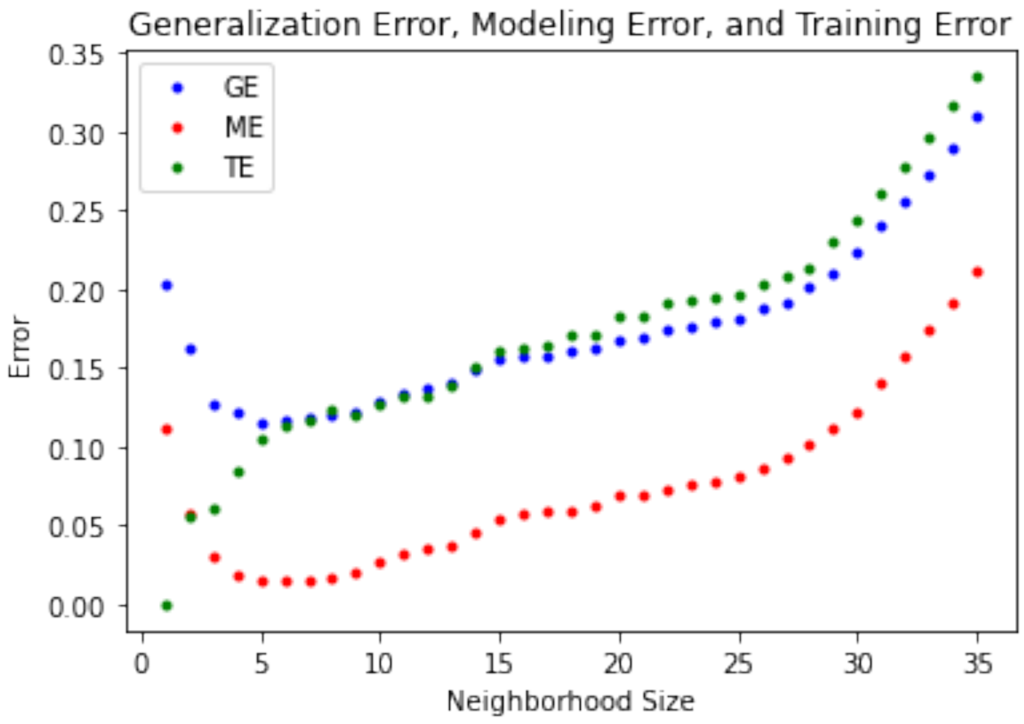

Figure 5. Visualization of the GE, ME, and TE plotted against neighborhood sizes

Right off the bat we can see that the training error is not a good estimator of model performance at all because we see at neighborhood size K=1, the model seemingly has very minimal, if any, error at all! This makes absolutely no sense, and is a clear indication that the model has very high variance is just over-fitting the data it's being trained on. When we look at the modeling and generalization error things start to get clearer, unfortunately we wouldn't be able to usually plot the modeling error in the real world because it requires knowing the actual function model which we usually will not have (otherwise we wouldn't even need to utilize machine learning). One thing to note is that the GE is very similar in terms of the error curve to the ME, and that means that the GE is effectively the next best evaluation metric to gauge model performance. When looking at both the GE and the ME we can see that the error starts off relatively high, ~0.20 for GE and 0.12 for ME, and then reaches its minima at K=5 neighborhood size before increasing. This is in-line with what was discussed from Figure 4 and how we can see that K=5 is the best neighborhood size to model this specific problem using KNNRegressor. The GE error at k=5 is ~0.12, whereas the ME error at k=5 is ~0.02. Note that to generate this plot, all K's in the range from 0 - 35 were used and their performance was evaluated, this includes K=1,2,5,35.

Conclusions

To conclude, in Figure 4 we went over the different outcomes from using KNN with k=1,2,5,35. It was found that K=1 and K=2 had high variance and low bias, and failed to properly fit the model. High variance is often found in models that are very complex, similar to what K=1 and K=2 are. When looking at K=5, we find a more optimal balance between bias and variance wherein the model does not overfit to the training data, and it performs well in mapping out the actual intended sin curve. K=5 has lower variance than K=1 and K=2 and higher bias, making it the optimal neighborhood size to use in this case. K=35 has very low variance, but extremely high bias. It fails to delineate any actual curve fitting, and is not a good model to use in this case.

Next in Figure 5 we analyzed the error curves for the TE, ME, and GE. Right away we noticed that the TE had 0 error at neighborhood size K=1, this makes no sense intuitively as we just visualized the outcome of the model when K was 1 and it was severely overfitting and failed to predict the curve properly. GE and ME were better estimates to use to evaluate the performance of a model, the ME however, is usually unavailable as it requires knowledge of the actual model function – something that isn't readily available in the real world. It was found that in both GE and ME there was a minimum at K=5, meaning that the error was the least in both cases at K=5, which is in-line with results observed from visualizing the models from Figure 4.

Lastly, I should note that the KNN function provided by the professor wasn't sufficient to generate the TE. I created my own version of the KNN function (using the code provided by the professor) to incorporate aspects needed in order to effectively calculate the TE. Please see the *Appendix* for further details, I added clear comments in the code before I made my own function and edited the doc-string of the function I made to explain more of what I added and changed.

References

None used, just followed the homework instruction and notes.

Appendix

```
# %% [markdown]
# # Assignment 2

# %% [markdown]
# Create a program to evaluate the Generalization Error (GE), 
Prediction Model Error (ME) and Training Error (TE) for the k-
nearest neighbors (KNN) learning approach. For doing so, compute 
the model considering neighborhood sizes from 1 to 35.

# %% [markdown]
# ### Imports:

# %%
from sklearn.neighbors import KNeighborsRegressor
import numpy as np

# %% [markdown]
# ### Built-in Custom Functions:

# %%
def KNN(N_size, x_train, Y_train, x_test):
    '''
    This function implements the KNN regression learning model. 
The required
    inputs are the following:
    -   N_size (integer): size of the neighborhood. 
Automatically reduced to
    training dataset size if greater than it.
    -   x_train (1-D list): list of x values related to the 
training data set.
    -   Y_train (1-D list): list of Y values related to the 
training data set.
    -   x_test (1-D list): list of x values related to the 
testing data set.

    The function outputs the following:
    -   Y_hat (1-D list): list containing the KNN regressed 
values for the 
    x_test data set according to the model training.
    '''

    N_size = np.minimum(len(x_train), N_size)

    x_i = [[x] for x in x_train]

    KNN = KNeighborsRegressor(N_size).fit(x_i, Y_train)

    x_i = [[x] for x in x_test]

    Y_hat = KNN.predict(x_i)

    return Y_hat


# %% [markdown]
# ## Solution Code:

# %% [markdown]
# ### Data Sets and Learning Model:
# 

# %% [markdown]
# #### Training Set:
# 
# With $N^{training} = 50$:
# 
# 
# - Generate $x_i$, $N^{training}$ uniformly separated data 
points between 0 and 1.
# 
# - Generate $n_i$, $N^{training}$ noise data points randomly 
distributed with 0 mean and 0.1 variance.
# 
# - Build the observed data model as: 
```

```
#
# $Y_i^{training} = f(x_i) + n_i$, with $\space i = 1 ... 
N^{training}$ and $f(x) = sin(2 \pi · x)$

# %%
# Defining function to generate a set of data
def gen_data(n_samples: int) -> np.ndarray:
    def func_x(input):
        return np.sin((2)*(np.pi)*(input))

    #x_i = np.linspace(0, 1, num=n_samples)

    x_i = []
    delx = 1/n_samples

    for x in range(n_samples):
        x_i.append(x*delx)
    

    #noise
    n_i = np.random.normal(loc=0, scale=np.sqrt(0.1), 
size=n_samples)

    #combining features + noise
    generated_samples = func_x(np.array(x_i)) + n_i


    #Dataset size
    print(f'Shape of generated labels: 
{np.shape(generated_samples)}\nShape of generated inputs: 
{np.shape(np.array(x_i))}')

    return np.array(x_i), generated_samples 

# %% [markdown]
# ### Generating training set (`n_samples` = 50)

# %%
X_train, y_train = gen_data(n_samples=50)

# %% [markdown]
# #### Testing Set:
#
# With $N^{testing} = 300$:
#
#
```

```
# - Follow the same previous steps, using $N^{testing}$ instead 
of $N^{training}$.

# %%
X_test, y_test = gen_data(n_samples=300)

# %%
print(f'y_train: {y_train}')
print('---------------------')
print(f'y_test: {y_test}')

# %% [markdown]
# #### Learning Model:
# 
# Use the K-Nearest Neighbors to evaluate its performance. Plot 
the model result for neighborhood sizes of 1, 5, 15, 25 and 40.

# %%
import matplotlib.pyplot as plt

# %%
N_sizes_plot = [1,2,5,35] #Plot these neighborhood sizes

legend_names=['K=1', 'K=2', 'K=5', 'K=35', 'Y_train']
for n in N_sizes_plot:
    y_hat = KNN(N_size=n, x_train=X_train, Y_train=y_train, 
x_test=X_test)
    plt.plot(X_test, y_hat)
    plt.legend(legend_names)

#plt.plot(X_train, y_train)
plt.title('Y_hat trained w/ KNN with differing neighborhood 
sizes')
plt.ylabel('Y')
plt.xlabel('X')
plt.show()


# %%
plt.plot(np.sin((2)*(np.pi)*(X_test)))
plt.title('Actual Function: sin(2*pi*x)')
plt.ylabel('Y')
plt.xlabel('X')

# %%
plt.plot(X_train, y_train)
```

```python
plt.plot(X_test, y_test, alpha=0.5)
plt.legend(['Training model', 'Testing model'])
plt.title('Training and Testing model based on f(x)')
plt.xlabel('X')
plt.ylabel('Y')

# %%
def plot_model_comparison(y_hat, k):
    plt.plot(X_test, np.sin(2*np.pi*X_test))
    plt.plot(X_test, y_hat)
    plt.plot(X_test, y_test, alpha=0.45)
    plt.title(f'Learning Model Comparison, K = {k}')
    plt.legend(['f(x)', 'KNN', 'y test'])
    plt.xlabel('X')
    plt.ylabel('Y')

for n in N_sizes_plot:
    y_hat = KNN(N_size=n, x_train=X_train, Y_train=y_train, 
x_test=X_test)
    plot_model_comparison(y_hat, k=n)
    plt.show()

# %% [markdown]
# Clearly, K = 5 fits it the best

# %% [markdown]
# ### Evaluation:

# %% [markdown]
# #### Error analysis

# %% [markdown]
# Note that I had to edit the KNN function provided above in 
order for the error analysis to work properly. There was 
currently an error where the previously provided KNN function 
fell short in calculating the training error because after 
regressing it would just predict on the testing set. However in 
the case of calculating TE we need the KNN to predict over the 
training set and that y_hat was used in the calculation of TE. I 
simply added an input argument `predict_test` which when `False` 
would use the training set to make y_hat predictions.

# %%
def KNN_edited(N_size, x_train, Y_train, x_test, 
predict_test:bool):
    '''
    This function implements the KNN regression learning model. 
The required
    inputs are the following:
    -   N_size (integer): size of the neighborhood. 
Automatically reduced to
    training dataset size if greater than it.
    -   x_train (1-D list): list of x values related to the 
training data set.
    -   Y_train (1-D list): list of Y values related to the 
training data set.
    -   x_test (1-D list): list of x values related to the 
testing data set.
    -   predict_test (boolean): True or False wether or not you 
want to make predictions on the testing set.
    If False it will default and make predictions on the 
training set (x_train)

    The function outputs the following:
    -   Y_hat (1-D list): list containing the KNN regressed 
values for the 
    x_test data set or x_train data setaccording to the model 
training.
    '''

    N_size = np.minimum(len(x_train), N_size)
    if predict_test: #If we want to make predictions on the 
x_test
        x_i = [[x] for x in x_train]
        KNN = KNeighborsRegressor(N_size).fit(x_i, Y_train)
        x_i = [[x] for x in x_test]
        Y_hat = KNN.predict(x_i)
    elif not predict_test: #If we want to make predictions on 
the x_train, this is used in the TE calculation
        x_i = [[x] for x in x_train]
        KNN = KNeighborsRegressor(N_size).fit(x_i, Y_train)
        Y_hat = KNN.predict(x_i)
    return Y_hat


# %%
N_test = 300
N_train = 50
N_sizes_plot = [1,2,5,35] #Plot these neighborhood sizes
f_x = np.sin(2.*np.pi*X_test)

for n in range(1,36):
    ge = 0
    me = 0
    y_hat = KNN_edited(N_size=n, x_train=X_train, 
Y_train=y_train, x_test=X_test, predict_test=True)
    for j in range(0,N_test):
        ge+=((y_test[j]-y_hat[j])**2) #Generalization error 
calculation
        me+=((f_x[j]-y_hat[j])**2) #Modeling error calculation

    me = me/N_test
    ge = ge/N_test
    te = 0

    y_hat_TE = KNN_edited(N_size=n, x_train=X_train, 
Y_train=y_train, x_test=X_test, predict_test=False)
    for k in range(0,N_train):
        te+=((y_train[k] - y_hat_TE[k])**2) #Training error 
calculation
        #print(np.shape(y_hat))
    te = te/N_train

    plt.plot(n, ge, '.b')
    plt.plot(n, me, '.r')
    plt.plot(n, te, '.g')
    plt.title('Generalization Error, Modeling Error, and 
Training Error')
    plt.ylabel('Error')
    plt.xlabel('Neighborhood Size')
    plt.legend(['GE', 'ME', 'TE'])
    #plt.show()
```