Nonparametric Regression with K-Nearest Neighbors

In [5]:
```
import numpy as np
import pandas as pd
from plotnine import *
```

Let's start with a simulated function which we want to estimate nonparametrically:

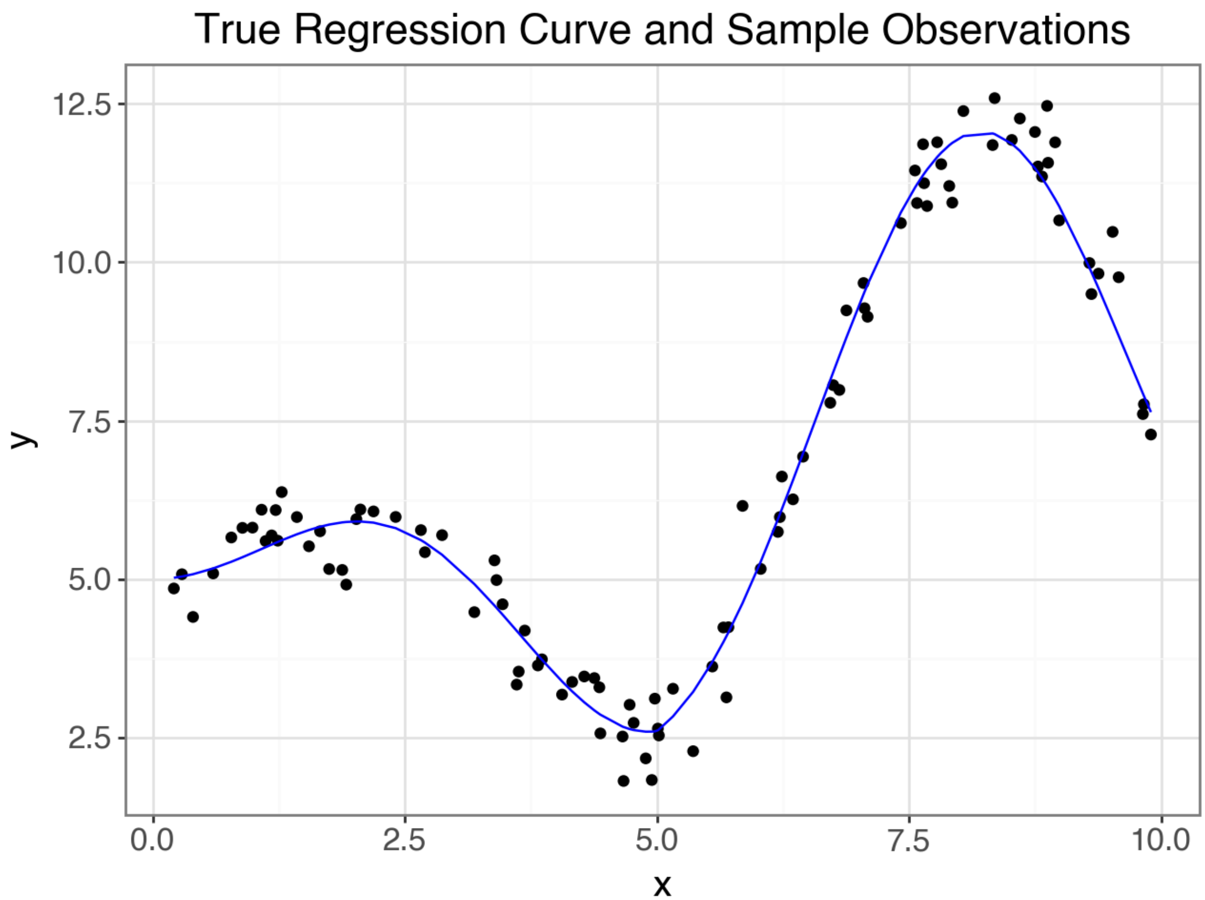
In [6]:
```
# Define the true regression function
def g(x):
    return np.maximum(x, 5) + 0.5 * x * np.sin(x)

# Generate the data
np.random.seed(1234)  # For reproducibility
x = np.arange(0, 10, 0.01)
x_sample = np.sort(np.random.choice(x, 100, replace=False))
y = g(x_sample) + np.random.normal(0, 0.5, size=100)

# Create dataframes to hold simulated data
data = pd.DataFrame({'x': x_sample, 'y': y})
data_nonoise = pd.DataFrame({'x': x_sample, 'g': g(x_sample)})
```

In [7]:
```
plot1 = (
    ggplot() +
    geom_point(data, aes(x='x', y='y'), color='black') +
    geom_line(data_nonoise, aes(x='x', y='g'), color='blue') +
    theme_bw() +
    labs(title='True Regression Curve and Sample Observations', y='y') +
    theme(
        plot_title=element_text(ha='center', size=16),
        axis_title=element_text(size=14),
        axis_text=element_text(size=12)
    )
)

plot1
```



Let's try to implement KNN regression from scratch:

For example, let's try to predict y when x = 2, using K=10 nearest neighbors...

Our first step will be to find the 10 nearest neighbors near x=2!

In [8]:
```
# Pick the x point of interest
x_point = 2

# Calculate distances from observations to x_point & sort them
dist = np.abs(data['x'] - x_point)
sorted_dist = dist.sort_values()

# We want the K smallest distances
K = 10
k_indices = sorted_dist.index[:K]
k_dists = sorted_dist.iloc[:K]

# Find out which points these correspond to
neighbor_x = data.loc[k_indices, 'x']
print("Neighbor x-values:", neighbor_x.values)
```

```
Neighbor x-values: [2.02 2.06 1.92 1.88 2.19 1.75 1.66 2.41 1.55 1.43]
```

Let's wrap this into a function which takes x and K as arguments so we can use it repeatedly and build onto it:

In [9]:
```python
def neighbors(x_point, K, xvar, data):
    # Calculate distances & sort them
    dist = np.abs(data[xvar] - x_point)
    sorted_dist = dist.sort_values()
    k_dists = sorted_dist[:K]

    # Find out which points these correspond to
    neighbor_ind = np.where(np.isin(dist, k_dists))[0]

    # Break ties by randomly subsetting down to K
    if len(neighbor_ind) != K:
        neighbor_ind = np.random.choice(neighbor_ind, K, replace=False)

    neighbor_x = data.iloc[neighbor_ind][xvar]

    # Return the indices and x-values for the K nearest neighbors
    out = {'ind': neighbor_ind, 'xvals': neighbor_x.values}
    return out
```

Before moving on, apply it to the situation before to sanity check that it works...:

In [10]:
```python
neighbors(x_point = 2, K = 10, xvar = 'x', data = data)
```

Out[10]:
```
{'ind': array([13, 14, 15, 16, 17, 18, 19, 20, 21, 22]),
 'xvals': array([1.43, 1.55, 1.66, 1.75, 1.88, 1.92, 2.02, 2.06, 2.19, 2.4
1])}
```

Ok cool, now let's use these neighbors to predict the y value!

In [11]:
```python
nearby_points_idx = neighbors(x_point = 2, K = 10, xvar = 'x', data = data)[
y_hat_knn = data.iloc[nearby_points_idx]['y'].mean()
y_hat_knn
```

Out[11]:
```
5.656884670423805
```

In [12]:
```python
def knn_predict(x_point, K, xvar, yvar, data):
    nearby_points_idx = neighbors(x_point, K, xvar, data)['ind']
    knn_pred = data.iloc[nearby_points_idx][yvar].mean()
    return knn_pred

# Example prediction at x = 2
prediction = knn_predict(2, K=10, xvar='x', yvar='y', data=data)
print(f"KNN Prediction at x=2: {prediction.round(4)}")
```

```
KNN Prediction at x=2: 5.6569
```

Predicting over a grid of x-values...

In [13]:
```python
# make a grid of x-values
x_grid = np.arange(0, 10, 0.1)
predictions = [knn_predict(x_point, K=10, xvar='x', yvar='y', data=data) for

pred_df = pd.DataFrame({'x': x_grid, 'y': predictions})
pred_df.head()
```

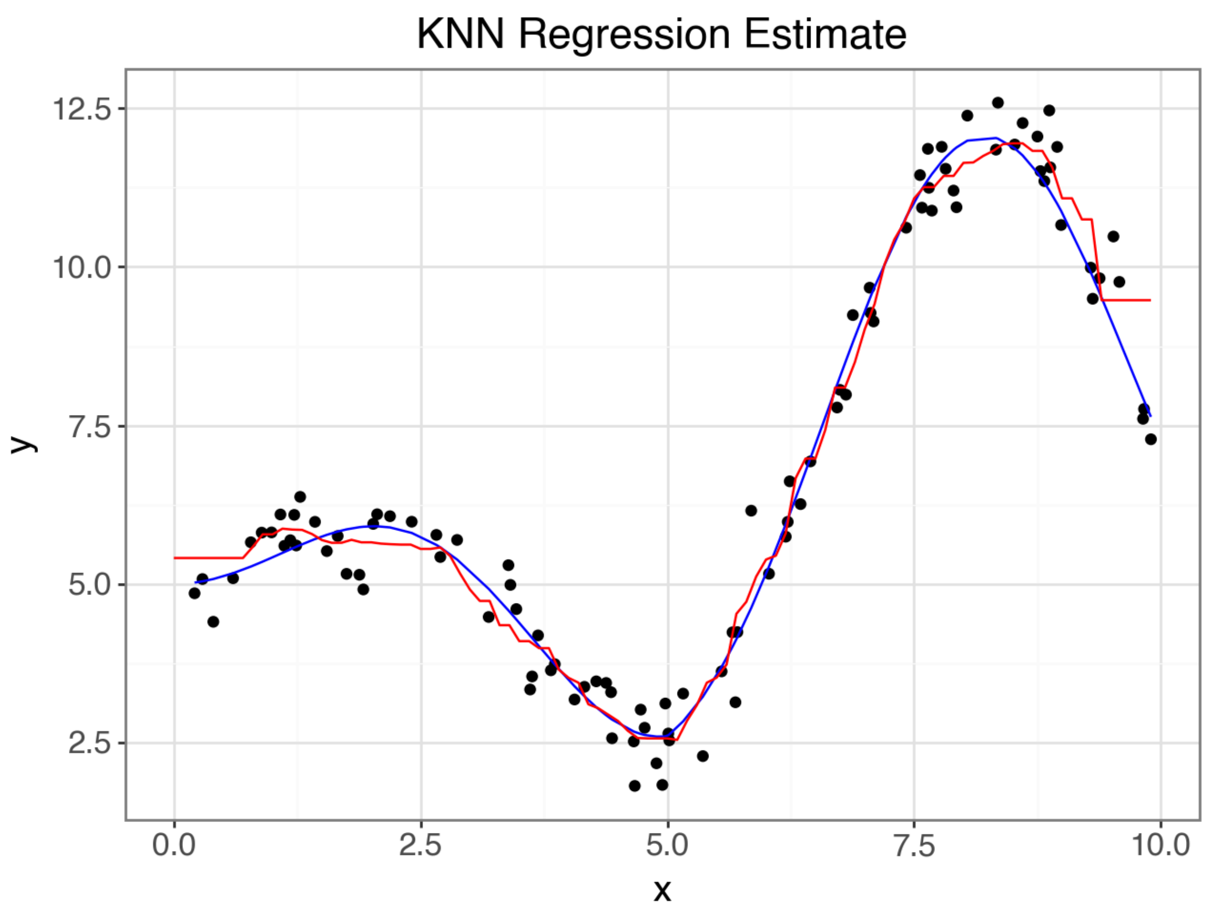
Out[13]:

	x	y
0	0.0	5.409209
1	0.1	5.409209
2	0.2	5.409209
3	0.3	5.409209
4	0.4	5.409209

And plotting, where red is the estimate and blue is the true function:

In [14]:

```
# Plot the data, true function, and KNN estimate
plot2 = (
    ggplot() +
    geom_point(data, aes(x='x', y='y'), color='black') +
    geom_line(data_nonoise, aes(x='x', y='g'), color='blue') +
    geom_line(pred_df, aes(x='x', y='y'), color='red') +
    theme_bw() +
    labs(title='KNN Regression Estimate', y='y') +
    theme(
        plot_title=element_text(ha='center', size=16),
        axis_title=element_text(size=14),
        axis_text=element_text(size=12)
    )
)
plot2
```



If we want to assess goodness of fit we can calculate the SSE:

In [15]:
```
# Calculate fitted values and SSE
fitted_values = [knn_predict(xi, K=10, xvar='x', yvar='y', data=data) for xi
sse = np.sum((data['y'] - fitted_values) ** 2)
print(f"SSE for K=10: {sse}")
```

```
SSE for K=10: 34.190330283485366
```

In [16]:
```
# Define the SSE function
def SSE(K, data):
    fitted_values = [knn_predict(xi, K=K, xvar='x', yvar='y', data=data) for
    sse = np.sum((data['y'] - fitted_values) ** 2)
    return sse

# Example SSE
sse_value = SSE(K=10, data=data)
print(f"SSE for K=10: {sse_value}")
```

```
SSE for K=10: 34.218171317054306
```

We expect these values to be the same... why is the SSE changing slightly each time we run it?

How to choose the best value of K?

We could perform data splitting and choose the value of K which minimizes the validation data set mean square error. Or we could do K-fold cross validation if we want to let each data point have a turn in the validation set.

K-fold Cross-Validation Idea:

- For a grid of K values, `K` in `K_grid`:

- For each observation i = 1,...,n:

1. Exclude the ith observation one at a time. This ith observation will serve as the "validation set."
2. Fit the K-Nearest Neighbor regression on the remaining $n-1$ observations (the training set).
3. Predict the y value for the ith data point.
4. Calculate squared prediction error and store it as $SSE_{(i)}$.

After you've done this for each i = 1,...,n, average the squared prediction errors. This is the cross-validated MSE:

$MSE_{CV}(K) = \frac{1}{n}\sum_{i=1}^{n} SSE_{(i)}$.

Choose the value of `K` in `K_grid` which minimizes $MSE_{CV}(K)$

Exercise:

1. Write a function which performs leave-one-out cross-validation.
2. Apply it to our dataset in this simulation. What is the optimal value of K? Call it K^*.
3. Calculate the SSE of the cross-validated KNN regression model with $K = K^*$. Compare it to the SSE for our initial choice of K=10. Does the tuning make a big difference in this case or were we close to correct with our initial hyperparameter choice?

Challenge:

- How can we extend the idea of KNN regression to the setting where we have multiple predictors, say X_1 and X_2? Describe the process in words and then write a function which implements this.

In [17]:
```
def LOOCV(K, data):
    n = len(data)
    errors = np.zeros(n)

    for i in range(n):
```

```
        # Exclude the ith observation
        train_data = data.drop(index=data.index[i]).reset_index(drop=True)

        # The x and y values of the left out point
        x_i = data.iloc[i]['x']
        y_i = data.iloc[i]['y']

        # Predict y_i using the model trained on train_data
        y_pred = knn_predict(x_i, K=K, xvar='x', yvar='y', data=train_data)

        # Compute squared error
        errors[i] = (y_pred - y_i) ** 2

    # Sum up squared errors
    loocv_error = np.sum(errors)/n
    return loocv_error
```

In [18]:
```
# Perform LOOCV for K=10
loocv_error = LOOCV(K=10, data=data)
print(f"LOOCV Error for K=10: {loocv_error}")
```

```
LOOCV Error for K=10: 0.4647813749582214
```

In [20]:
```
K_grid = np.arange(1, 100, 1)
```

In [21]:
```
loocv_K_MSE = [LOOCV(K=k, data=data) for k in K_grid]
```

In [22]:
```
kstar_idx = np.where(np.isin(loocv_K_MSE, np.min(loocv_K_MSE)))
```

In [23]:
```
kstar = K_grid[kstar_idx]
kstar #K* = 3
```

Out[23]:
```
array([3])
```

In [24]:
```
LOOCV(K=3, data=data) #better than K = 10
```

Out[24]:
```
0.3276057972997473
```

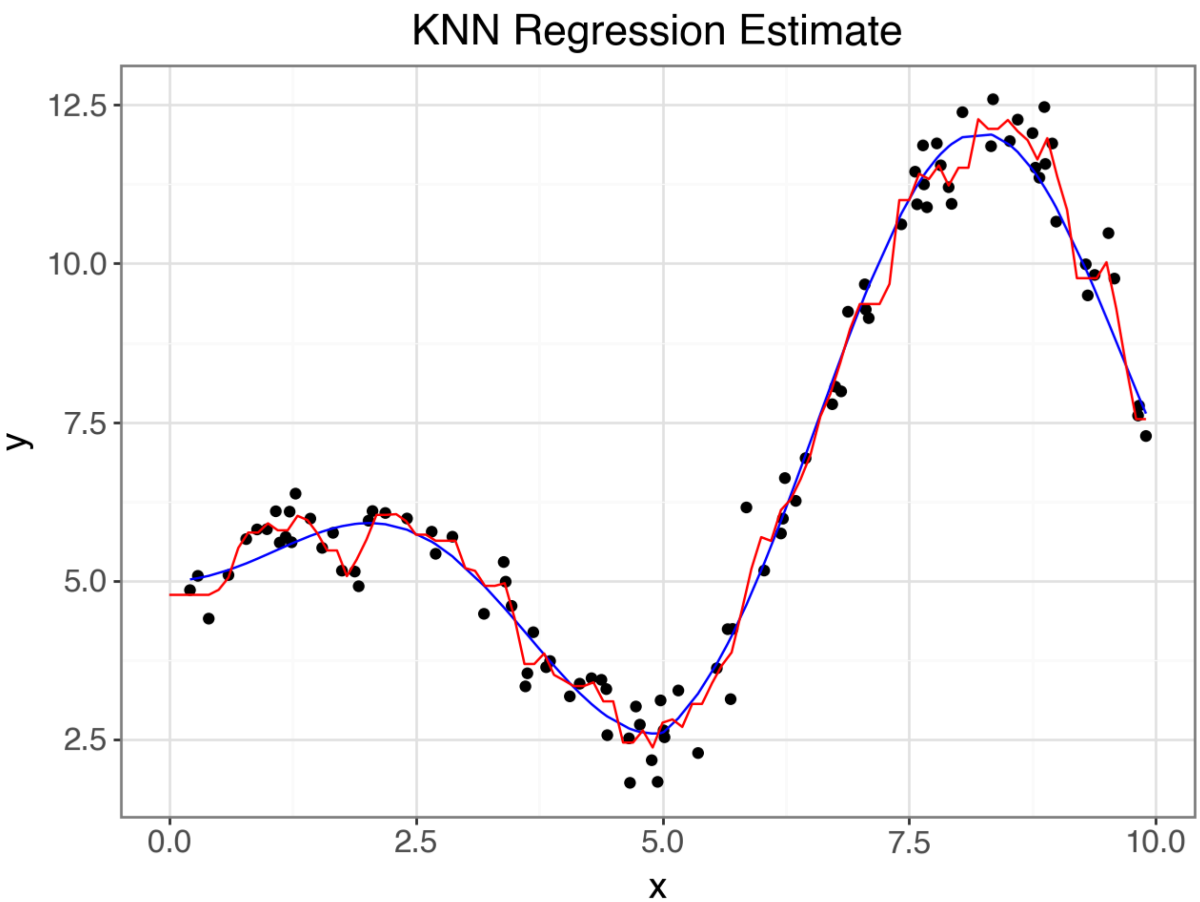
In [25]:
```
x_grid = np.arange(0, 10, 0.1)
predictions3 = [knn_predict(x_point, K=3, xvar='x', yvar='y', data=data) for

pred_df3 = pd.DataFrame({'x': x_grid, 'y': predictions3})
pred_df3.head()
```

Out[25]:

	x	y
0	0.0	4.778216
1	0.1	4.778216
2	0.2	4.778216
3	0.3	4.778216
4	0.4	4.778216

In [26]:
```
# Plot the data, true function, and KNN estimate
plot3 = (
    ggplot() +
    geom_point(data, aes(x='x', y='y'), color='black') +
    geom_line(data_nonoise, aes(x='x', y='g'), color='blue') +
    geom_line(pred_df3, aes(x='x', y='y'), color='red') +
    theme_bw() +
    labs(title='KNN Regression Estimate', y='y') +
    theme(
        plot_title=element_text(ha='center', size=16),
        axis_title=element_text(size=14),
        axis_text=element_text(size=12)
    )
)
plot3
```



It's a slight improvement over the previous but overall its pretty close. It is smoother as well, which may be a benefit.

In [ ]: