Lesson 5
Evaluating and Comparing Learners

Evaluation and Why Less Is More

- » The biggest danger in Machine Learning is thinking your model is a "genius" when it has actually just memorized the answers.
- » The Two Pitfalls
  - » **Limited Capacity**: If a model is too simple (underfitting), it doesn't have the "brainpower" to learn the patterns in the data.
  - » **Distraction by Noise**: If a model is too complex, it starts treating random "noise" or coincidences in the data as important rules (overfitting).

Pitfall	**Cause**	**Model State**	**Performance**
Limited Capacity	Model is too simple	**Underfitting**	Bad on Train & Test
Distraction by Noise	Model is too complex	**Overfitting**	Great on Train / Bad on Test

Evaluation and Why Less Is More (continued)

Example: KNN

1. Underfitting with k-NN (High Bias)

- **How it happens:** This occurs when you set k to a **very high number** (e.g., k=100).
- **The Effect:** The model averages so many neighbors that it ignores local variations. It essentially treats everyone as "average".
- **The Result:** The model performs poorly on the training data and also poorly on the testing data.

2. Overfitting with k-NN (High Variance)

- **How it happens:** This occurs when you set k to a **very low number**, especially k=1.
- **The Effect:** At k=1, the model perfectly "memorizes" every outlier or error in the training set. If one patient has an unusual health profile due to a measurement error, the model will follow that error exactly.
- **The Result:** The model will have **0% error on training data** but will perform very poorly on new testing data because it hasn't learned to generalize.

Evaluation and Why Less Is More (continued)

Example: Naïve Bayes

1. Underfitting with Naïve Bayes (High Bias)

- **How it happens:** This often occurs if you have too few features or if the features you chose have no statistical relationship with the outcome.
- **The Effect:** Imagine building a Spam Filter using only **one** feature: Does the email contain the word "Free"? Many legitimate emails (newsletters, receipts) use the word "Free," and many spam emails don’t.
  - The model is too simple; it "under-learns" the complexity of spam, leading to high error rates on both training and testing data.
- **The Result:** This is a state of High Bias, where the model's assumptions are too strong.

Evaluation and Why Less Is More (continued)

Example: Naïve Bayes

2. Overfitting with Naïve Bayes (High Variance)

- **How it happens:** This usually happens when you have a very small dataset with too many features.
- **The Effect:** You train a Spam Filter on only 5 emails. By chance, two of the spam emails were sent at 3:04 AM.
    - The Naive Bayes model might calculate a 100% probability that any email sent at 3:04 AM is spam.
    - It has "memorized" a random coincidence (noise) rather than a general pattern.
- **The Result:** This is **High Variance**. The model performs perfectly on those 5 emails, but fails when it sees an email from a friend sent at 3:04 AM.

Evaluation and Why Less Is More (continued)

» To avoid teaching to the test data, the following three steps are used:
  - ✓Split the data into separate training and testing datasets.
  - ✓Learn on the training data.
  - ✓Evaluate on the testing data.

- **The "Why Less is More"**
  - **For k-NN:** If you decrease k too much (k=1), you increase complexity and risk **Overfitting** (High Variance).
  - **For Naive Bayes:** If you set α too high, you "wash out" the data's signals, leading to **Underfitting** (High Bias).
- Finding the balance between these is the goal of the **Selection Phase** using your **Validation Data**.

Practice program using “mtcars” dataset

Parameters Vs. Hyperparameters

Algorithm	Parameters (Learned from Data)	Hyperparameters (Set by the Human)
k-Nearest Neighbors (k-NN)	**None.** k-NN is a "non-parametric" model. It stores the training data points themselves rather than learning specific weights or values.	**k (Number of Neighbors):** Controls the complexity. Small k can lead to overfitting; large k can lead to underfitting.
Naive Bayes	**Probabilities:** The model calculates the frequency of each feature given a class (e.g., How often does the word "Free" appear in Spam emails?).	**Smoothing (α)**(e.g., Laplace Smoothing). A value added to counts to prevent "zero-probability" errors for words not seen in training.

(Re)Sampling: Making More from Less

- 1 Cross-Validation
- 2 Stratification
- 3 Repeated Train-Test Splits
- 4 A Better Way and Shuffling
- 5 Leave-One-Out Cross-Validation

(Re)Sampling: Making More from Less (continued)

- A single train-test split provides and determines both the data to train from and the testing environment.
- It is a simple method to turn one dataset into many.

1.Cross-Validation

- Cross-validation (CV) is a resampling procedure to evaluate machine learning models on a limited data sample.
- The general name for CV techniques is k-fold cross-validation.
- The default scoring for regressors is **r^2**.

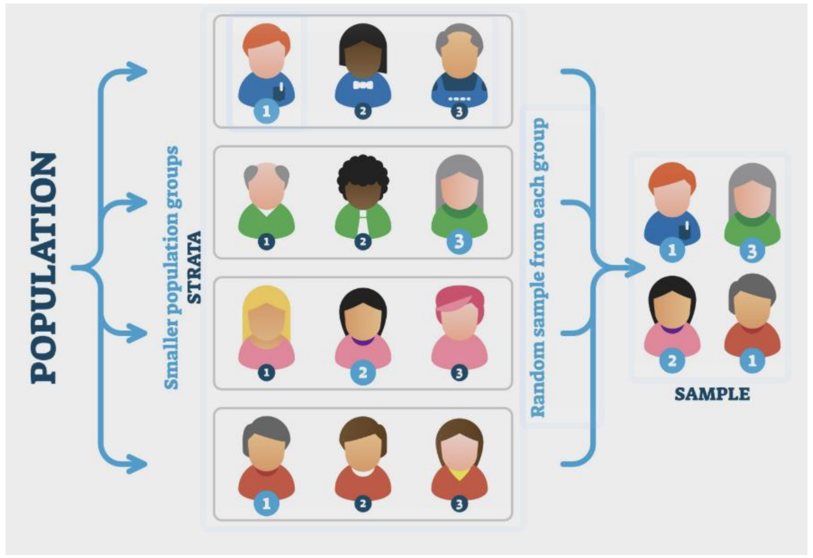
- **Simple Example**: In 5-Fold CV, you have 5 groups.
- Iterate: You run the training process 5 times.
  - In the first round, you train on groups 1, 2, 3, and 4, and test on group 5.
  - In the second round, you train on groups 1, 2, 3, and 5, and test on group 4.
  - You keep rotating until every group has been the "Test Set" exactly once.
- Average: You take the results (like MSE or Accuracy) from all 5 rounds and average them to get a final score.

2. Stratification

» Stratification ensures that your training and testing sets have the same **proportion** of data as the original set.

» It is particularly useful with limited data or classes that are poorly represented in the dataset.

» **Why?** It prevents the model from getting a "biased" test (e.g., testing a diabetes model on only healthy patients).



3. Repeated Train-Test Splits

- Repeated train-test splits provide a distribution of the evaluation measure, whether it is accuracy, root-mean-squared-error, or something else.
- Instead of one fixed split, you randomly shuffle and split the data many times and average the results.

4. A Better Way and Shuffling

» This emphasizes that **shuffling** is mandatory before splitting. If your data is ordered (e.g., all the "Spam" emails are at the top and "Inbox" emails are at the bottom), a split without shuffling will result in a model that only sees one type of data.

» Sklearn wrapped the process up in a single stand-alone function to make it precise.

» **ShuffleSplit** relies on a Python generator to produce one split after another.

5. Leave-One-Out CV (LOOCV)

- LOOCV is a deterministic evaluation method.
- This is the most extreme version of Cross-Validation. If you have N data points, you train on N-1 points and test on exactly one point. You repeat this N times.
- It is useful for comparing and testing correctness of learning algorithms.
- It can be expensive to run LOOCV because it requires to train the model once for each left-out example.
- It gives a relatively unbiased estimate of the real error rate.

Bias and Variance

- Variance of the Data
- Variance of the Model
- Bias of the Model

Bias and Variance

- In the world of machine learning, accuracy is everything.
- One can try to make the model more accurate by tuning and tweaking the parameters, but are never able to make it 100% accurate.
- A source of error is the interaction between the training data and the learner.

Variance of the Data

» **Variance of the Data** measures how spread out the individual data points are from their average (mean). It tells you the "diversity" or "volatility" of your information.

How it Affects Machine Learning

- **The "Noise" Problem:** High variance in data often means there is more "noise." If your data is very spread out and messy, a model might struggle to find the true signal, leading to the **overfitting** issues we discussed earlier.
- **Feature Scaling:** If one feature has a variance of 1000 (like annual income) and another has a variance of 0.5 (like height in meters), many models (like **k-NN**) will mistakenly think the income is "more important" just because the numbers are more spread out. This is why we **normalize** data.

Variance of the Model

» The way models vary due to the random selection of the data we train on is called the variance of the model.

The "Memorizer": Imagine a student who doesn't understand math but has a **photographic memory**.

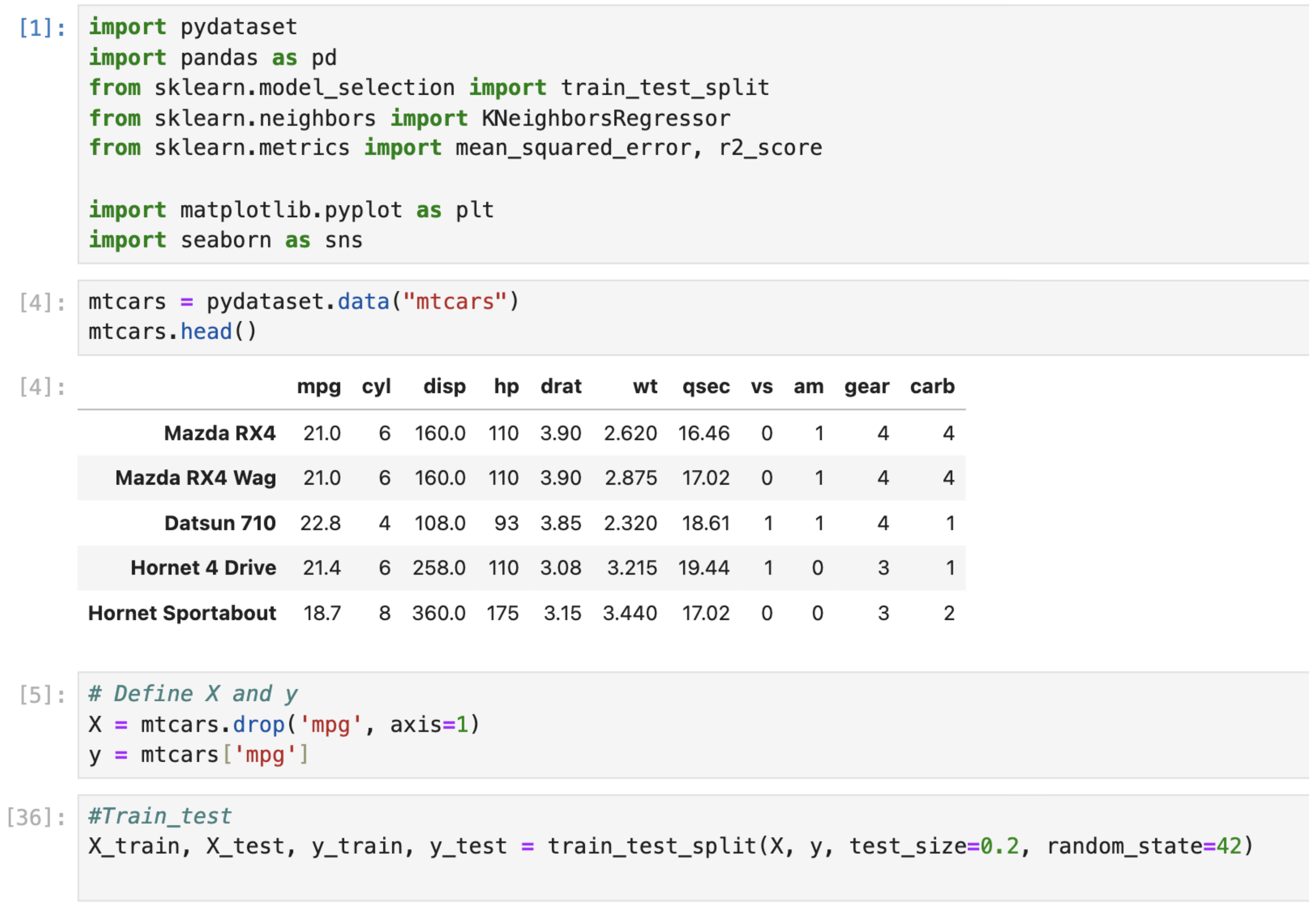
- If you give them Practice Test A, they memorize every single answer.
- If you give them Practice Test B (with different numbers), they fail because they only learned the "random" specific numbers from Test A, not the actual rules of math.

The student’s "answer key" changes drastically depending on which practice test you gave them. That **change** is what we call **Variance**.

Bias of the Model

- **Bias** is the error that occurs when a model makes oversimplified assumptions about the data. Instead of looking at the actual complexity of the relationship, the model forces a simple rule onto it.
- A high bias means the prediction will be inaccurate.
- A low bias and high variance problem is called overfitting.

Program – Evaluating a model



```
[1]: import pydataset
     import pandas as pd
     from sklearn.model_selection import train_test_split
     from sklearn.neighbors import KNeighborsRegressor
     from sklearn.metrics import mean_squared_error, r2_score

     import matplotlib.pyplot as plt
     import seaborn as sns
```

```
[4]: mtcars = pydataset.data("mtcars")
     mtcars.head()
```

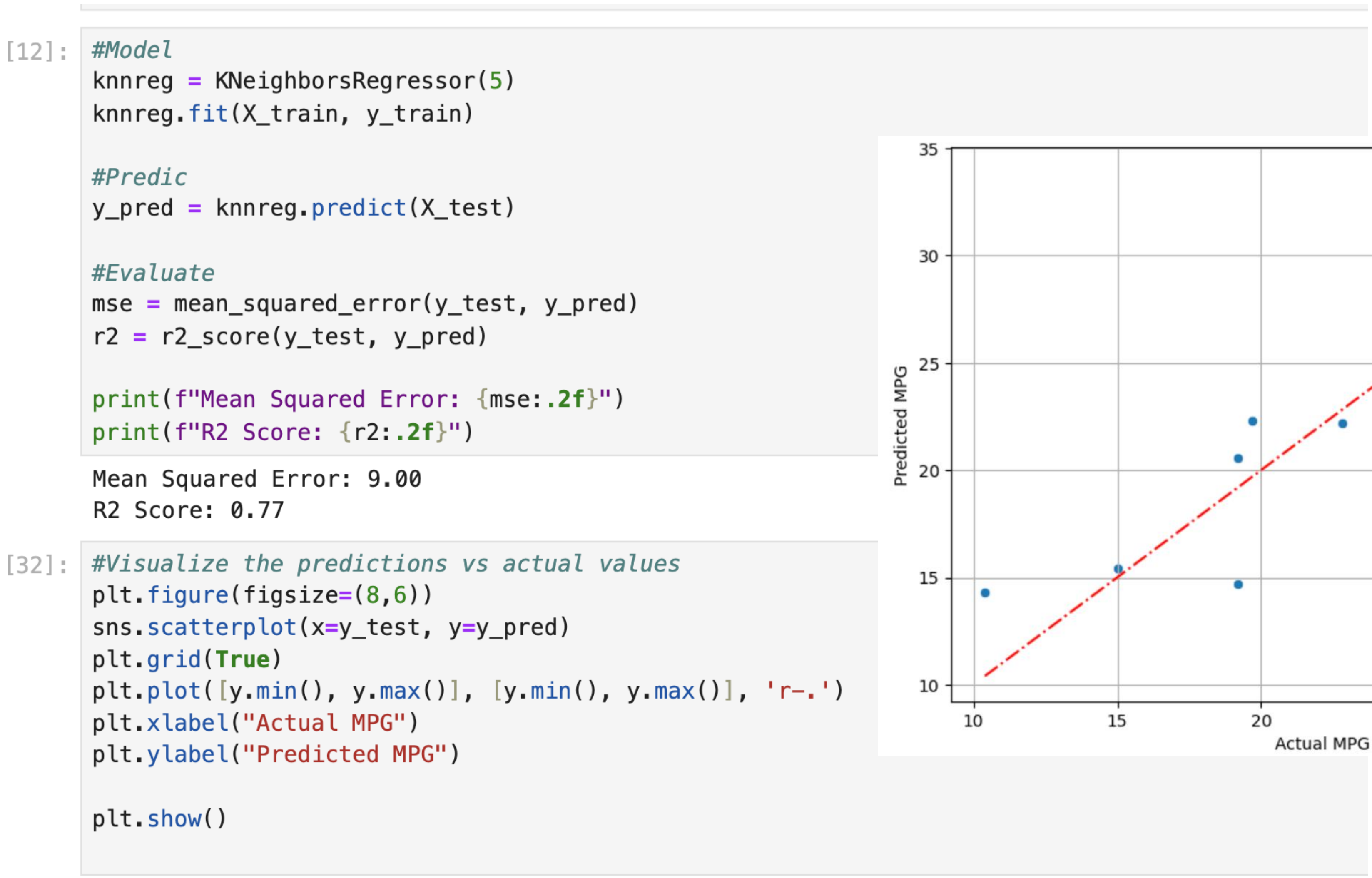
[4]:

	mpg	cyl	disp	hp	drat	wt	qsec	vs	am	gear	carb
Mazda RX4	21.0	6	160.0	110	3.90	2.620	16.46	0	1	4	4
Mazda RX4 Wag	21.0	6	160.0	110	3.90	2.875	17.02	0	1	4	4
Datsun 710	22.8	4	108.0	93	3.85	2.320	18.61	1	1	4	1
Hornet 4 Drive	21.4	6	258.0	110	3.08	3.215	19.44	1	0	3	1
Hornet Sportabout	18.7	8	360.0	175	3.15	3.440	17.02	0	0	3	2

```
[5]: # Define X and y
     X = mtcars.drop('mpg', axis=1)
     y = mtcars['mpg']
```

```
[36]: #Train_test
      X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
```

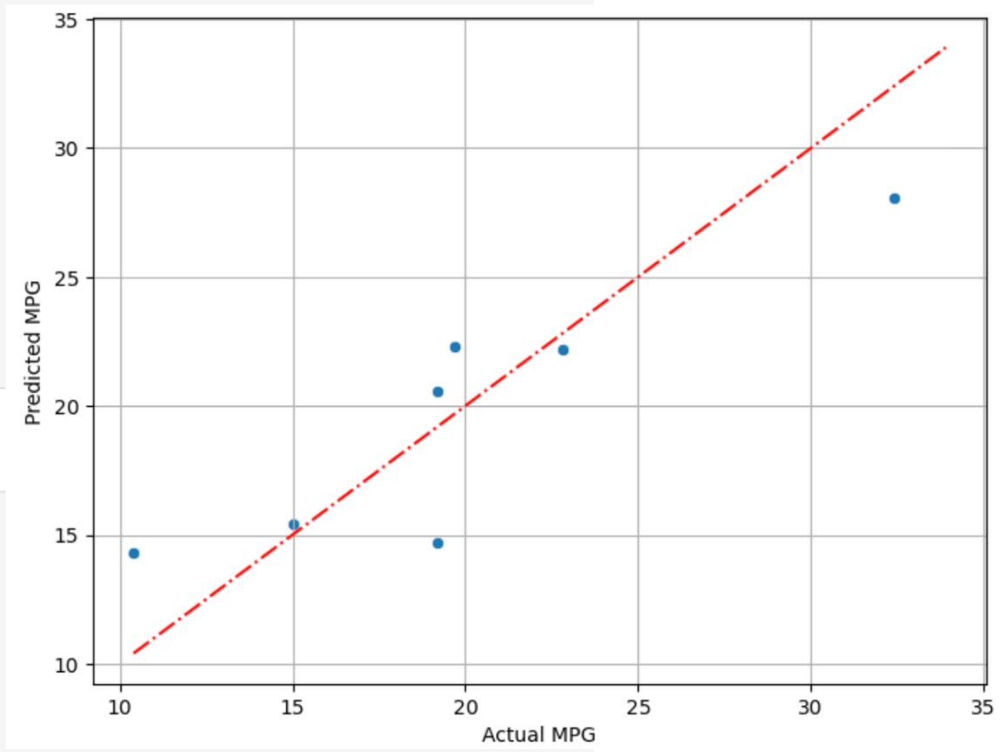
Program – Evaluating a model



```
[12]: #Model
      knnreg = KNeighborsRegressor(5)
      knnreg.fit(X_train, y_train)

      #Predic
      y_pred = knnreg.predict(X_test)

      #Evaluate
      mse = mean_squared_error(y_test, y_pred)
      r2 = r2_score(y_test, y_pred)

      print(f"Mean Squared Error: {mse:.2f}")
      print(f"R2 Score: {r2:.2f}")
```

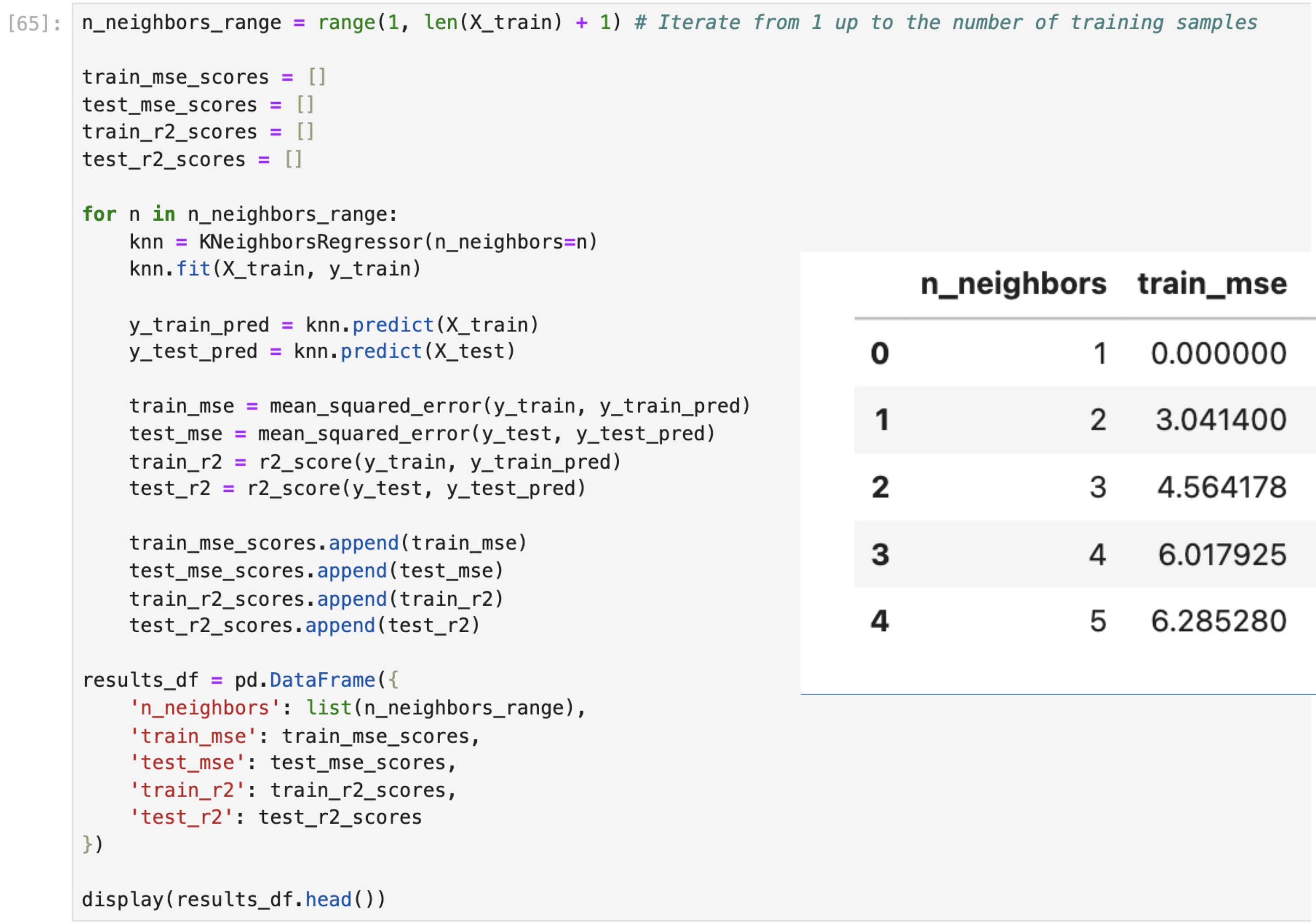
```
Mean Squared Error: 9.00
R2 Score: 0.77
```

```
[32]: #Visualize the predictions vs actual values
      plt.figure(figsize=(8,6))
      sns.scatterplot(x=y_test, y=y_pred)
      plt.grid(True)
      plt.plot([y.min(), y.max()], [y.min(), y.max()], 'r-.')
      plt.xlabel("Actual MPG")
      plt.ylabel("Predicted MPG")

      plt.show()
```

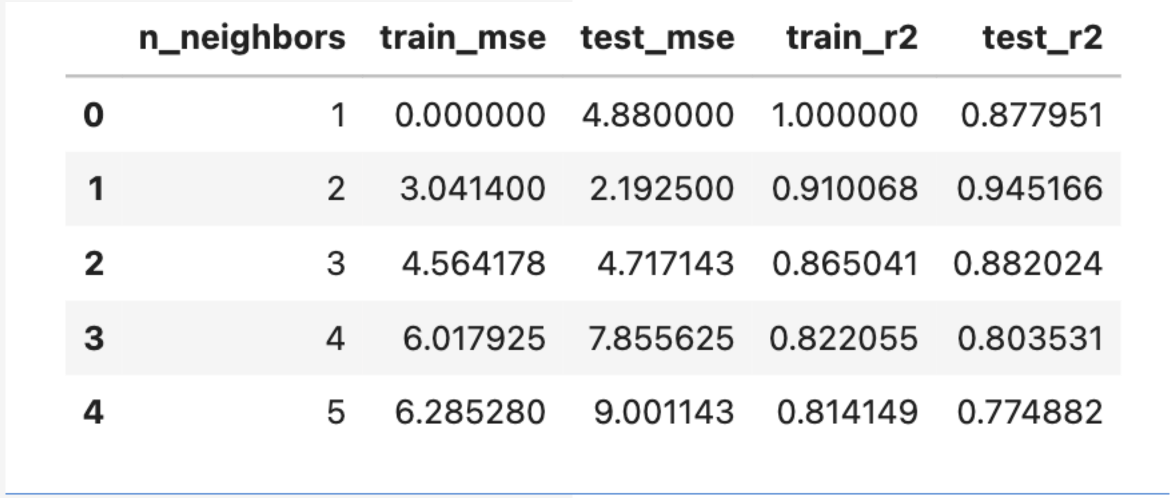


Program – Evaluating a model

Analyze Model fitness

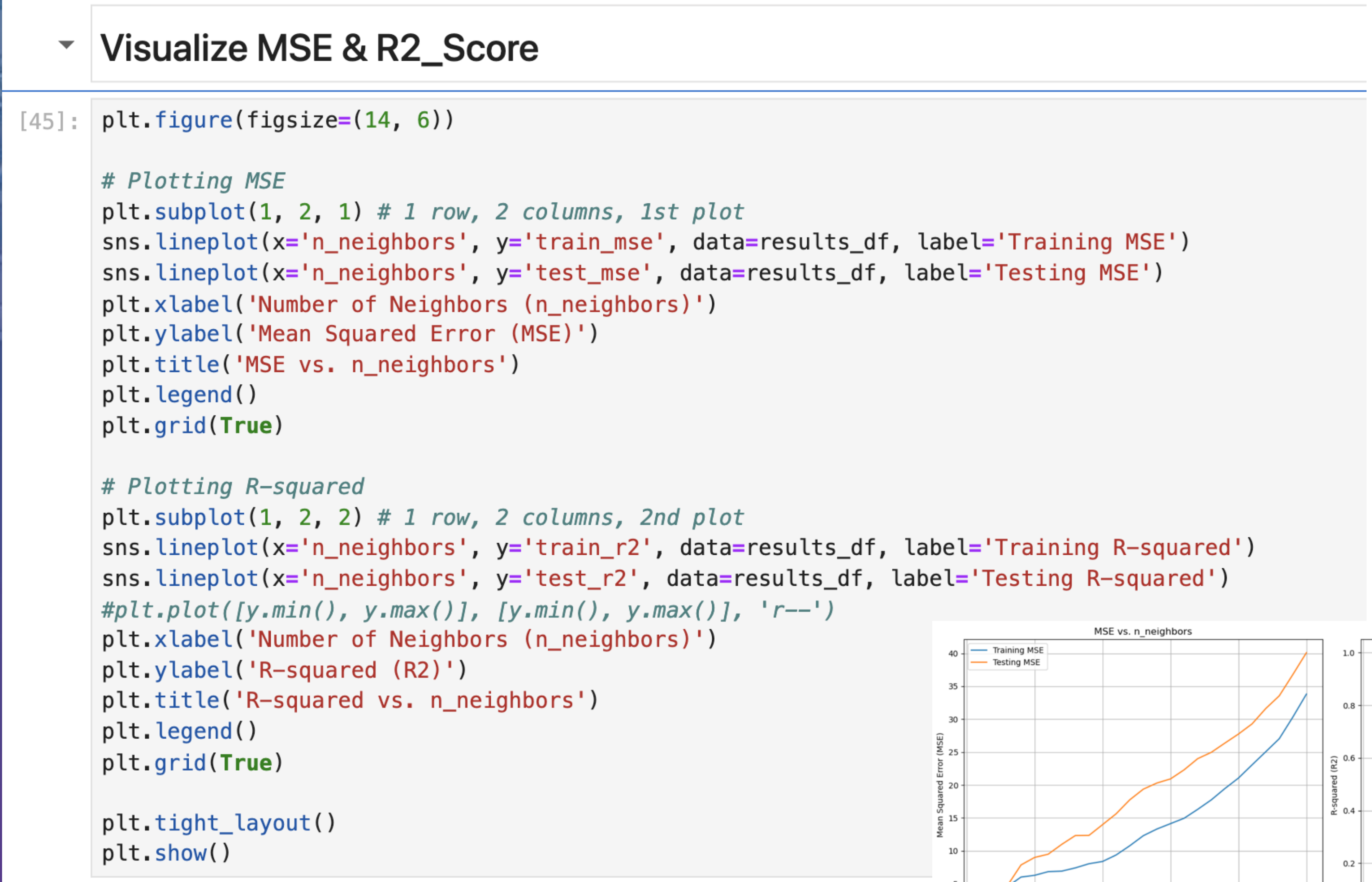


```
[65]: n_neighbors_range = range(1, len(X_train) + 1) # Iterate from 1 up to the number of training samples

      train_mse_scores = []
      test_mse_scores = []
      train_r2_scores = []
      test_r2_scores = []

      for n in n_neighbors_range:
          knn = KNeighborsRegressor(n_neighbors=n)
          knn.fit(X_train, y_train)

          y_train_pred = knn.predict(X_train)
          y_test_pred = knn.predict(X_test)

          train_mse = mean_squared_error(y_train, y_train_pred)
          test_mse = mean_squared_error(y_test, y_test_pred)
          train_r2 = r2_score(y_train, y_train_pred)
          test_r2 = r2_score(y_test, y_test_pred)

          train_mse_scores.append(train_mse)
          test_mse_scores.append(test_mse)
          train_r2_scores.append(train_r2)
          test_r2_scores.append(test_r2)

      results_df = pd.DataFrame({
          'n_neighbors': list(n_neighbors_range),
          'train_mse': train_mse_scores,
          'test_mse': test_mse_scores,
          'train_r2': train_r2_scores,
          'test_r2': test_r2_scores
      })

      display(results_df.head())
```



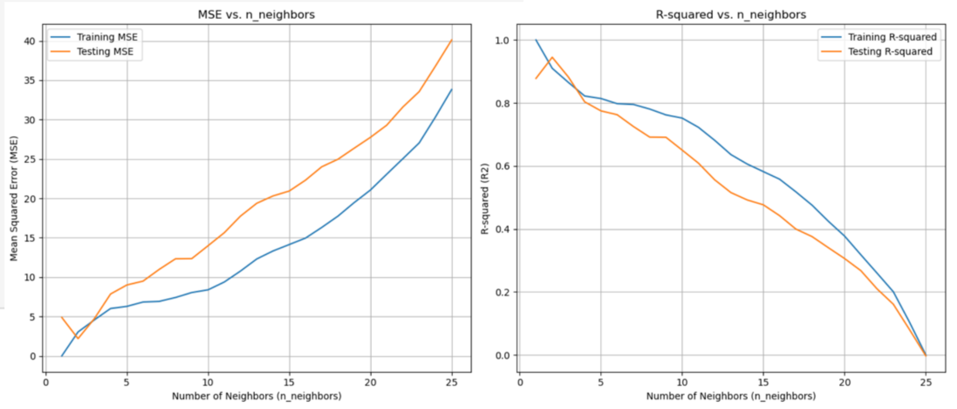
	n_neighbors	train_mse	test_mse	train_r2	test_r2
0	1	0.000000	4.880000	1.000000	0.877951
1	2	3.041400	2.192500	0.910068	0.945166
2	3	4.564178	4.717143	0.865041	0.882024
3	4	6.017925	7.855625	0.822055	0.803531
4	5	6.285280	9.001143	0.814149	0.774882

Program – Evaluating a model

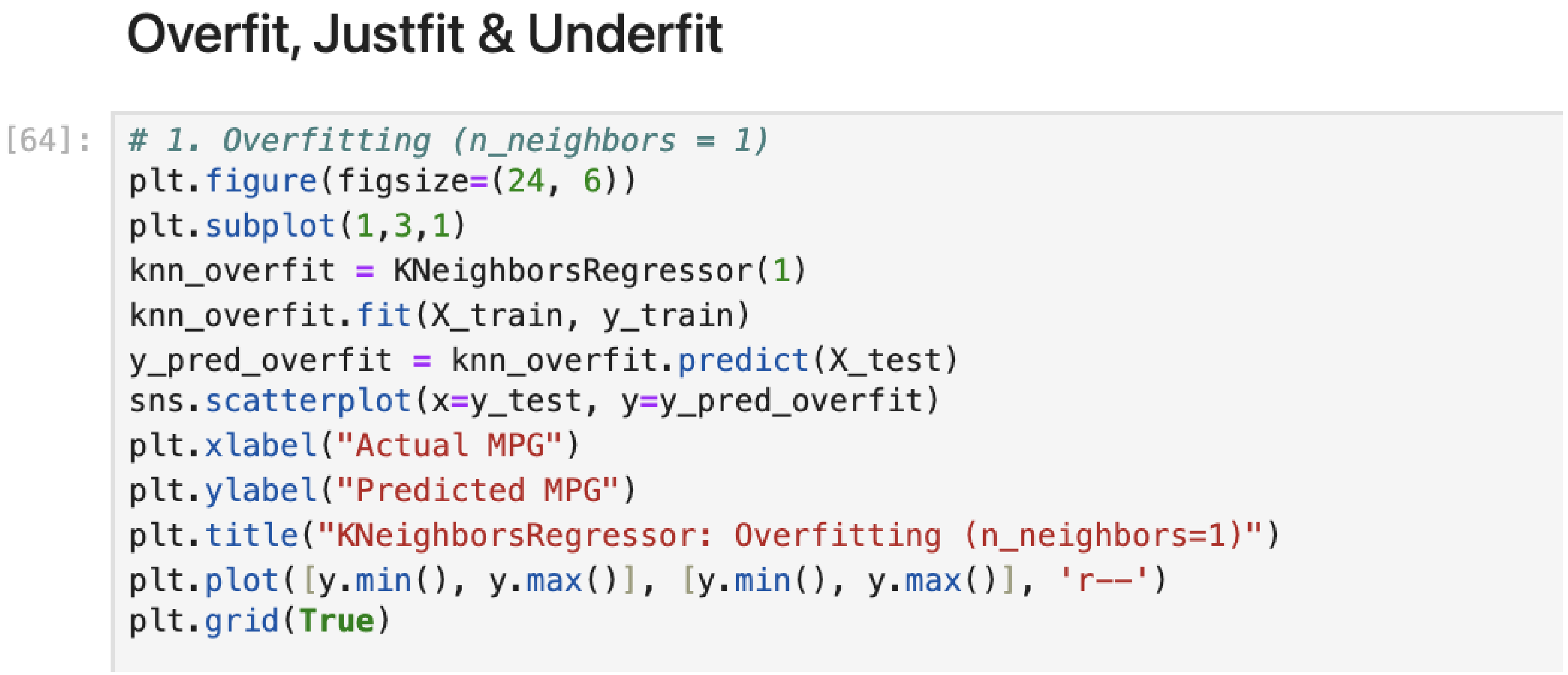


Visualize MSE & R2_Score

```
[45]: plt.figure(figsize=(14, 6))

      # Plotting MSE
      plt.subplot(1, 2, 1) # 1 row, 2 columns, 1st plot
      sns.lineplot(x='n_neighbors', y='train_mse', data=results_df, label='Training MSE')
      sns.lineplot(x='n_neighbors', y='test_mse', data=results_df, label='Testing MSE')
      plt.xlabel('Number of Neighbors (n_neighbors)')
      plt.ylabel('Mean Squared Error (MSE)')
      plt.title('MSE vs. n_neighbors')
      plt.legend()
      plt.grid(True)

      # Plotting R-squared
      plt.subplot(1, 2, 2) # 1 row, 2 columns, 2nd plot
      sns.lineplot(x='n_neighbors', y='train_r2', data=results_df, label='Training R-squared')
      sns.lineplot(x='n_neighbors', y='test_r2', data=results_df, label='Testing R-squared')
      #plt.plot([y.min(), y.max()], [y.min(), y.max()], 'r--')
      plt.xlabel('Number of Neighbors (n_neighbors)')
      plt.ylabel('R-squared (R2)')
      plt.title('R-squared vs. n_neighbors')
      plt.legend()
      plt.grid(True)

      plt.tight_layout()
      plt.show()
```



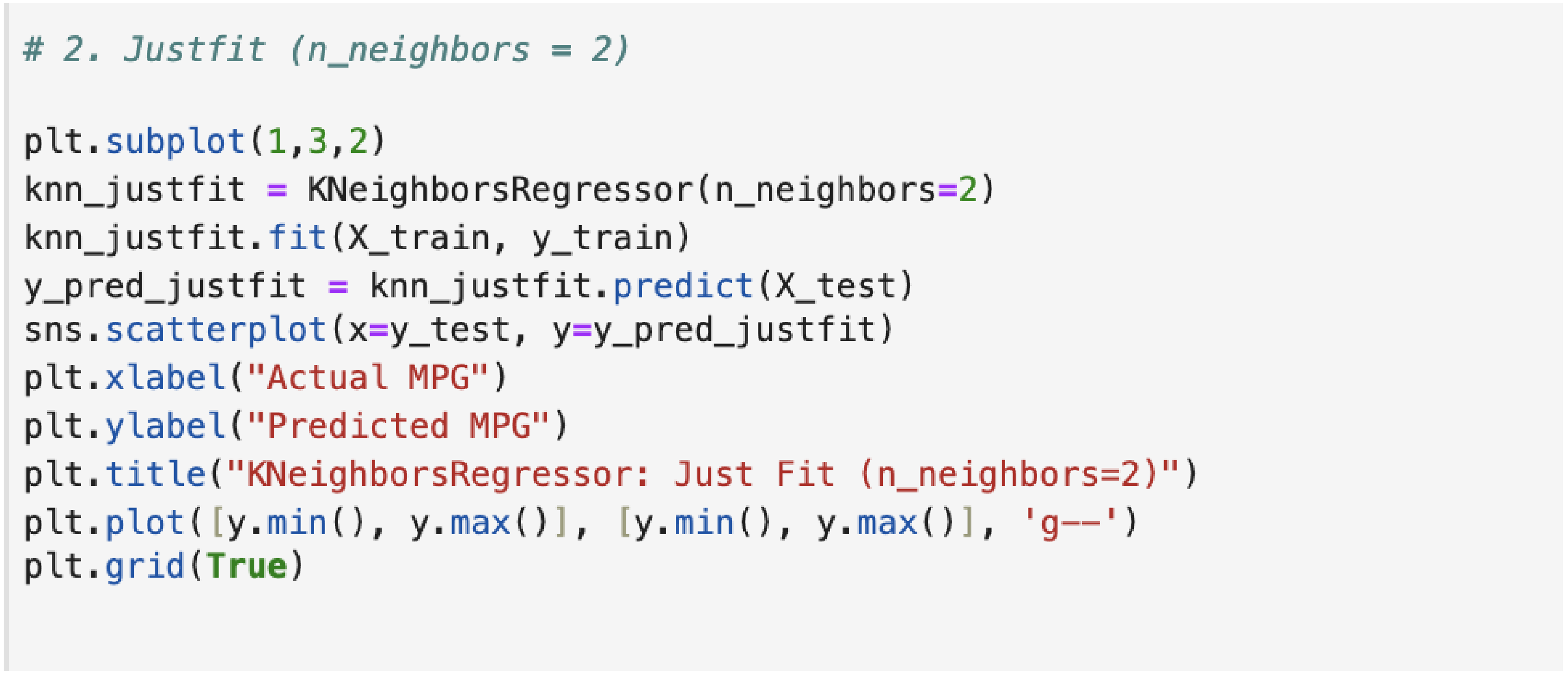
Program – Evaluating a model



Overfit, Justfit & Underfit

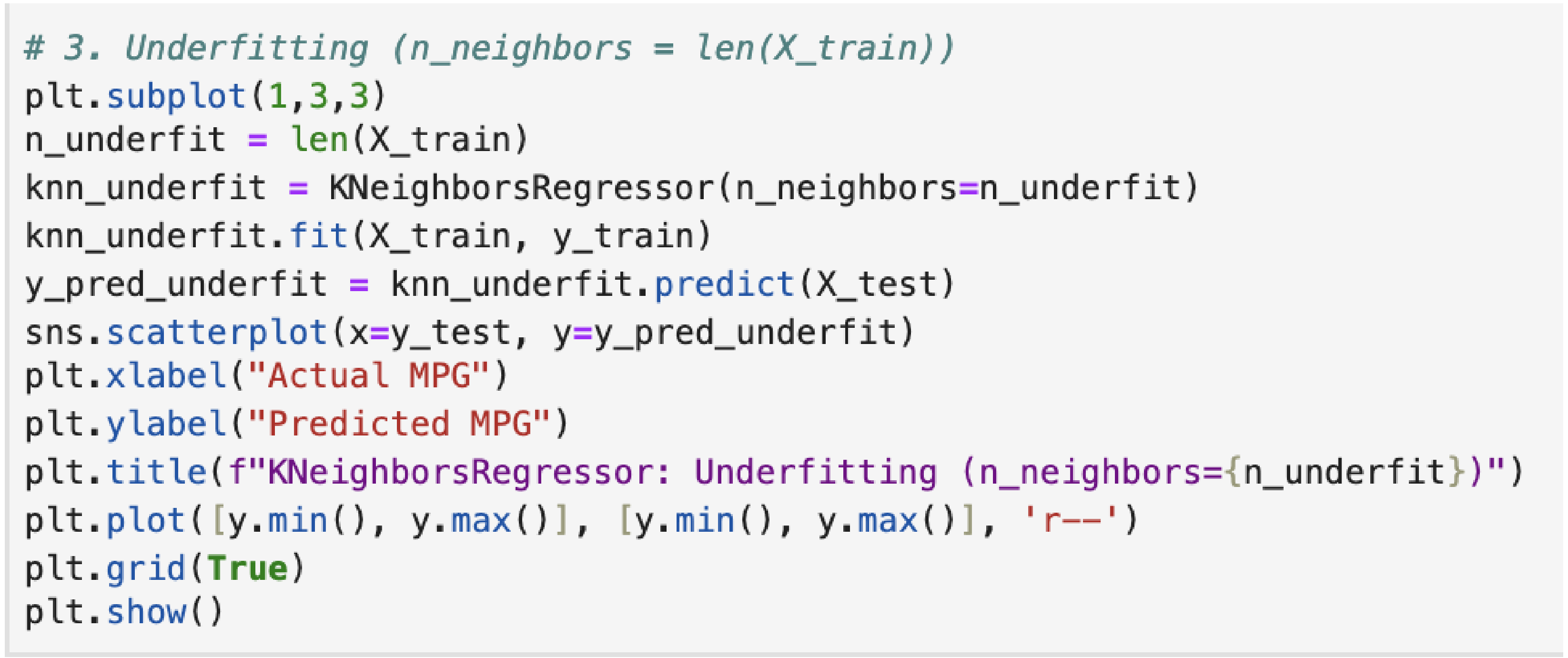
[64]:
```python
# 1. Overfitting (n_neighbors = 1)
plt.figure(figsize=(24, 6))
plt.subplot(1,3,1)
knn_overfit = KNeighborsRegressor(1)
knn_overfit.fit(X_train, y_train)
y_pred_overfit = knn_overfit.predict(X_test)
sns.scatterplot(x=y_test, y=y_pred_overfit)
plt.xlabel("Actual MPG")
plt.ylabel("Predicted MPG")
plt.title("KNeighborsRegressor: Overfitting (n_neighbors=1)")
plt.plot([y.min(), y.max()], [y.min(), y.max()], 'r--')
plt.grid(True)
```

Program – Evaluating a model



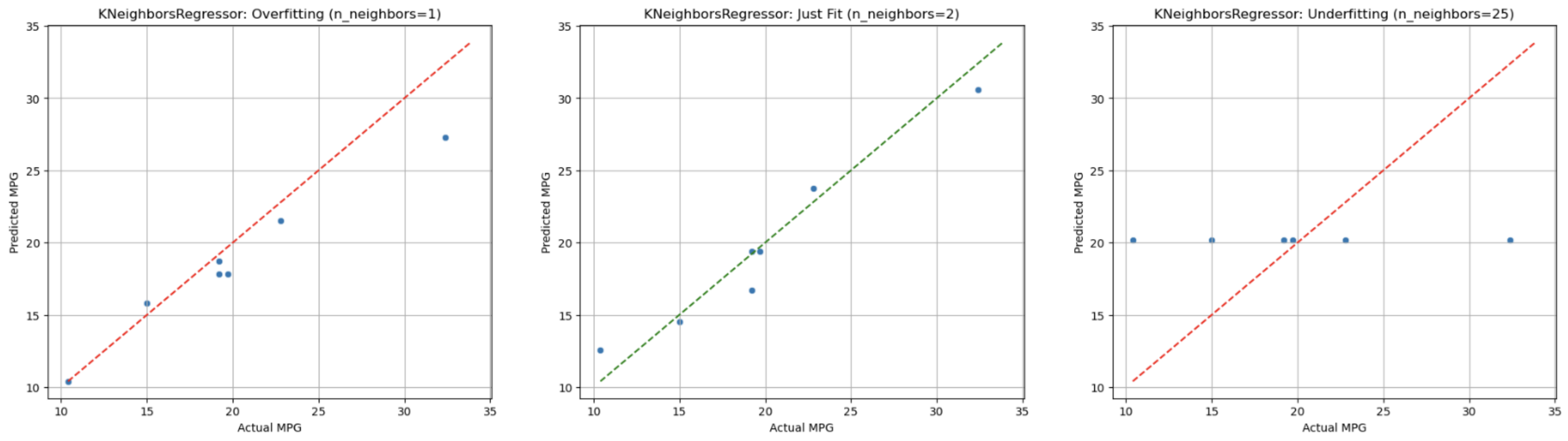
```python
# 2. Justfit (n_neighbors = 2)

plt.subplot(1,3,2)
knn_justfit = KNeighborsRegressor(n_neighbors=2)
knn_justfit.fit(X_train, y_train)
y_pred_justfit = knn_justfit.predict(X_test)
sns.scatterplot(x=y_test, y=y_pred_justfit)
plt.xlabel("Actual MPG")
plt.ylabel("Predicted MPG")
plt.title("KNeighborsRegressor: Just Fit (n_neighbors=2)")
plt.plot([y.min(), y.max()], [y.min(), y.max()], 'g--')
plt.grid(True)
```

Program – Evaluating a model



```python
# 3. Underfitting (n_neighbors = len(X_train))
plt.subplot(1,3,3)
n_underfit = len(X_train)
knn_underfit = KNeighborsRegressor(n_neighbors=n_underfit)
knn_underfit.fit(X_train, y_train)
y_pred_underfit = knn_underfit.predict(X_test)
sns.scatterplot(x=y_test, y=y_pred_underfit)
plt.xlabel("Actual MPG")
plt.ylabel("Predicted MPG")
plt.title(f"KNeighborsRegressor: Underfitting (n_neighbors={n_underfit})")
plt.plot([y.min(), y.max()], [y.min(), y.max()], 'r--')
plt.grid(True)
plt.show()
```

Program – Evaluating a model


KNeighborsRegressor: Overfitting (n_neighbors=1)
KNeighborsRegressor: Just Fit (n_neighbors=2)
KNeighborsRegressor: Underfitting (n_neighbors=25)
Predicted MPG
Actual MPG