Sales Prediction

1. Reading and Understanding the Data

In [1]:
```
# Supress Warnings

import warnings
warnings.filterwarnings('ignore')

# Import the numpy and pandas package

import numpy as np
import pandas as pd

# Data Visualisation
import matplotlib.pyplot as plt
import seaborn as sns
```

In [3]:
```
advertising = pd.DataFrame(pd.read_csv("advertising.csv"))
advertising.head()
```

Out[3]:

	TV	Radio	Newspaper	Sales
0	230.1	37.8	69.2	22.1
1	44.5	39.3	45.1	10.4
2	17.2	45.9	69.3	12.0
3	151.5	41.3	58.5	16.5
4	180.8	10.8	58.4	17.9

2. Data Inspection

In [4]:
```
advertising.shape
```

Out[4]: (200, 4)

In [5]:
```
advertising.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 200 entries, 0 to 199
Data columns (total 4 columns):
 #   Column     Non-Null Count  Dtype  
---  ------     --------------  -----  
 0   TV         200 non-null    float64
 1   Radio      200 non-null    float64
 2   Newspaper  200 non-null    float64
 3   Sales      200 non-null    float64
dtypes: float64(4)
memory usage: 6.4 KB
```

In [6]:
```
advertising.describe()
```

Out[6]:

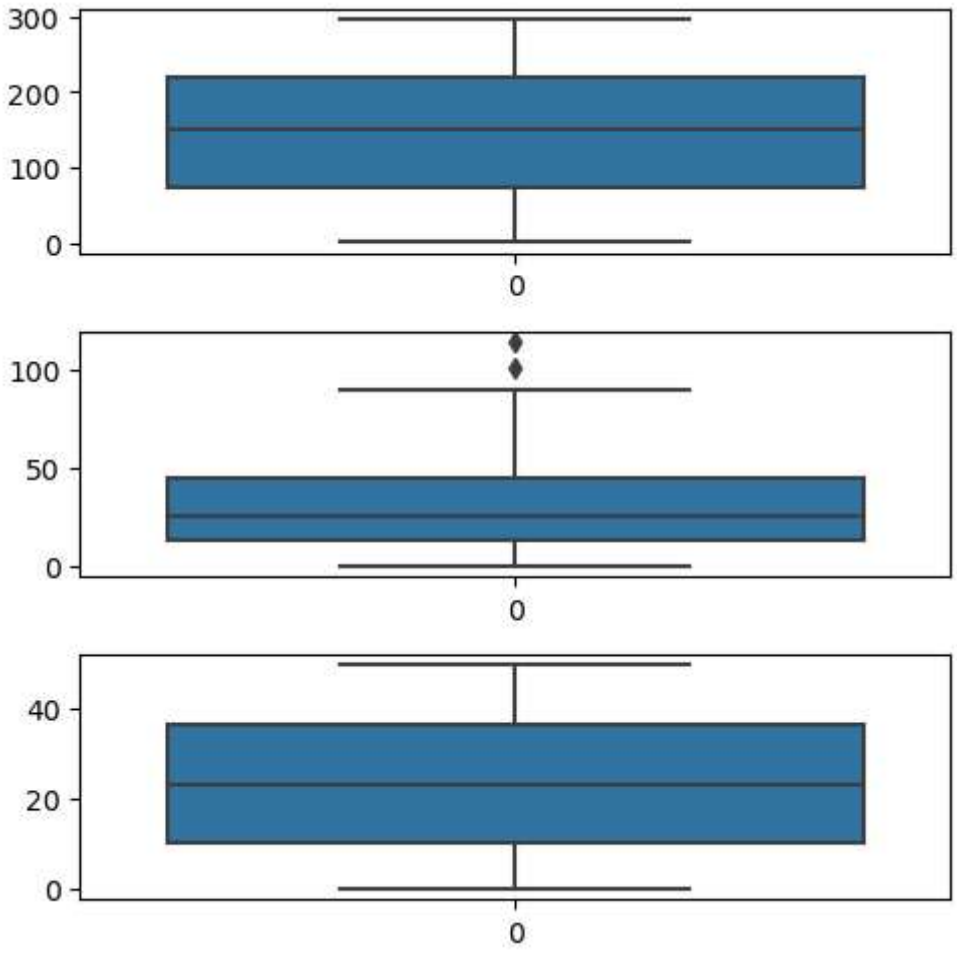
	TV	Radio	Newspaper	Sales
count	200.000000	200.000000	200.000000	200.000000
mean	147.042500	23.264000	30.554000	15.130500
std	85.854236	14.846809	21.778621	5.283892
min	0.700000	0.000000	0.300000	1.600000
25%	74.375000	9.975000	12.750000	11.000000
50%	149.750000	22.900000	25.750000	16.000000
75%	218.825000	36.525000	45.100000	19.050000
max	296.400000	49.600000	114.000000	27.000000

3. Data Cleaning

In [8]:
```
# Checking Null values
advertising.isnull().sum()*100/advertising.shape[0]
# There are no NULL values in the dataset, hence it is clean.
```

Out[8]:
```
TV           0.0
Radio        0.0
Newspaper    0.0
Sales        0.0
dtype: float64
```

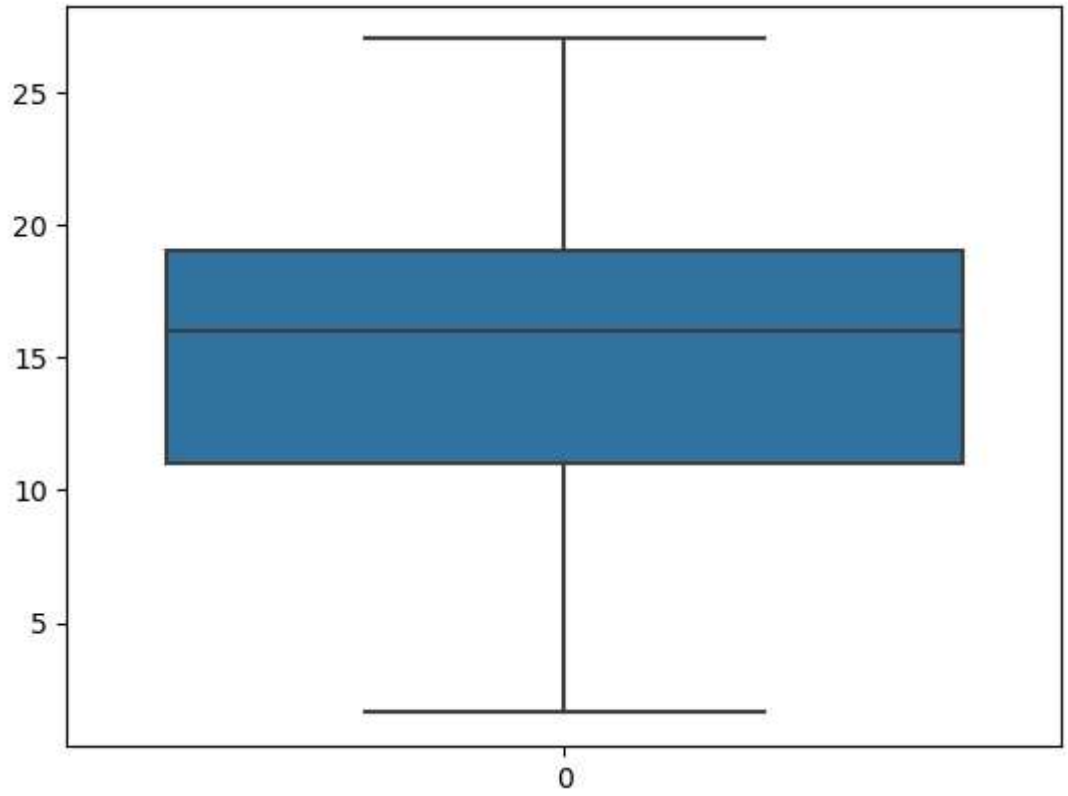
In [9]:
```
# Outlier Analysis
fig, axs = plt.subplots(3, figsize = (5,5))
plt1 = sns.boxplot(advertising['TV'], ax = axs[0])
plt2 = sns.boxplot(advertising['Newspaper'], ax = axs[1])
plt3 = sns.boxplot(advertising['Radio'], ax = axs[2])
plt.tight_layout()
```



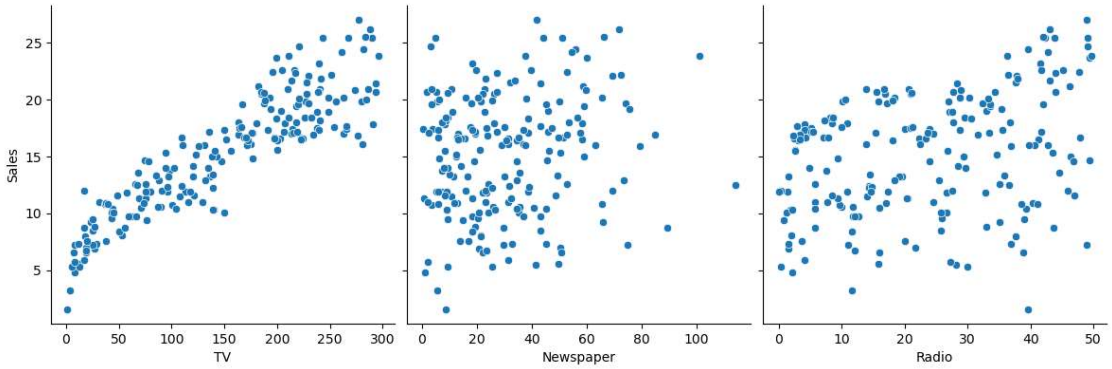
In [10]:
```
# There are no considerable outliers present in the data.
```

4. Exploratory Data Analysis

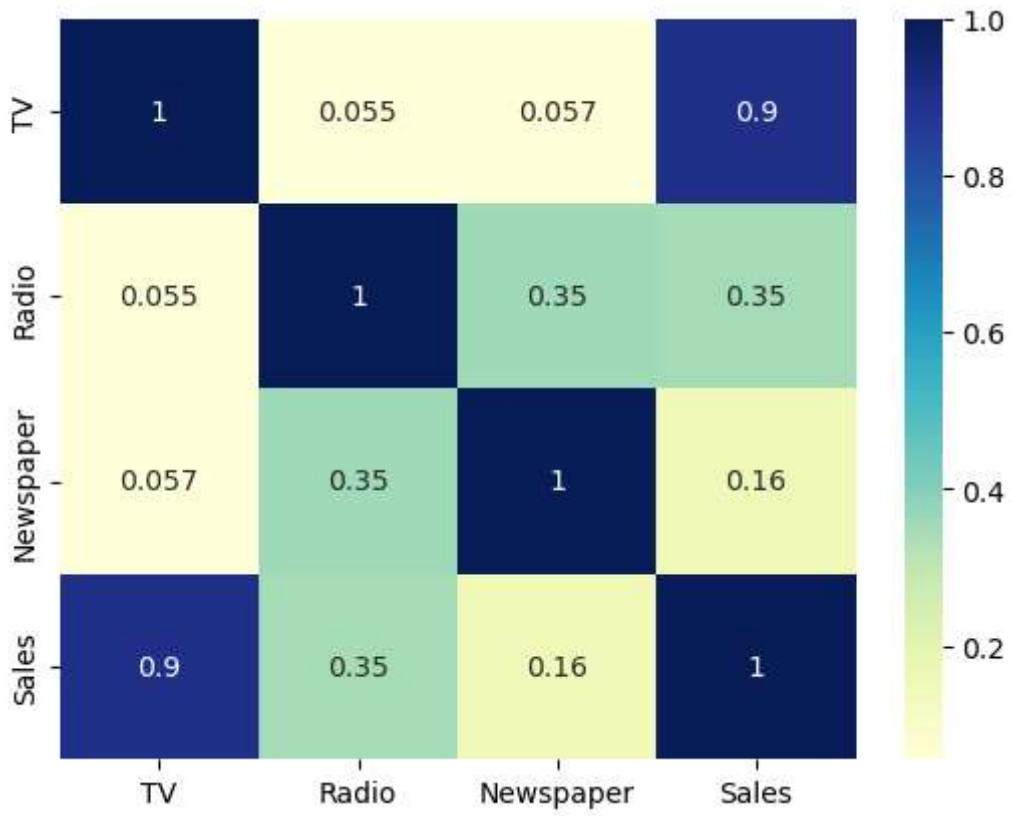
In [11]:
```
sns.boxplot(advertising['Sales'])
plt.show()
```



In [12]:
```
# Let's see how Sales are related with other variables using scatter plot
sns.pairplot(advertising, x_vars=['TV', 'Newspaper', 'Radio'], y_vars='Sa
plt.show()
```



In [13]:
```
# Let's see the correlation between different variables.
sns.heatmap(advertising.corr(), cmap="YlGnBu", annot = True)
plt.show()
```



As is visible from the pairplot and the heatmap, the variable `TV` seems to be most correlated with `Sales`. So let's go ahead and perform simple linear regression using `TV` as our feature variable.

5. Model Building

Performing Simple Linear Regression

Equation of linear regression
$y = c + m_1 x_1 + m_2 x_2 + \ldots + m_n x_n$

- y is the response
- c is the intercept
- m_1 is the coefficient for the first feature
- m_n is the coefficient for the nth feature

In our case:

$y = c + m_1 \times TV$

The m values are called the model **coefficients** or **model parameters**.

Generic Steps in model building using `statsmodels`

We first assign the feature variable, `TV`, in this case, to the variable `X` and the response variable, `Sales`, to the variable `y`.

In [14]:
```
X = advertising['TV']
y = advertising['Sales']
```

Train-Test Split

You now need to split our variable into training and testing sets. You'll perform this by importing `train_test_split` from the `sklearn.model_selection` library. It is usually a good practice to keep 70% of the data in your train dataset and the rest 30% in your test dataset

In [15]:
```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, train_size = 0.
```

In [16]:
```
# Let's now take a look at the train dataset

X_train.head()
```

Out[16]:
```
74     213.4
3      151.5
185    205.0
26     142.9
90     134.3
Name: TV, dtype: float64
```

In [17]:
```
y_train.head()
```

Out[17]:
```
74     17.0
3      16.5
185    22.6
26     15.0
90     14.0
Name: Sales, dtype: float64
```

Building a Linear Model

You first need to import the `statsmodel.api` library using which you'll perform the linear regression.

In [18]:
```
import statsmodels.api as sm
```

By default, the `statsmodels` library fits a line on the dataset which passes through the origin. But in order to have an intercept, you need to manually use the `add_constant` attribute of `statsmodels` . And once you've added the constant to your `X_train` dataset, you can go ahead and fit a regression line using the `OLS` (Ordinary Least Squares) attribute of `statsmodels` as shown below

In [19]:
```
# Add a constant to get an intercept
X_train_sm = sm.add_constant(X_train)

# Fit the resgression line using 'OLS'
lr = sm.OLS(y_train, X_train_sm).fit()
```

In [20]:
```
# Print the parameters, i.e. the intercept and the slope of the regressio
lr.params
```

Out[20]:
```
const    6.948683
TV       0.054546
dtype: float64
```

In [21]:
```
# Performing a summary operation lists out all the different parameters o
print(lr.summary())
```

```
                            OLS Regression Results
==============================================================================
Dep. Variable:                  Sales   R-squared:                       0.816
Model:                            OLS   Adj. R-squared:                  0.814
Method:                 Least Squares   F-statistic:                     611.2
Date:                Tue, 14 Nov 2023   Prob (F-statistic):           1.52e-52
Time:                        18:26:34   Log-Likelihood:                -321.12
No. Observations:                 140   AIC:                             646.2
Df Residuals:                     138   BIC:                             652.1
Df Model:                           1
Covariance Type:            nonrobust
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
const          6.9487      0.385     18.068      0.000       6.188       7.709
TV             0.0545      0.002     24.722      0.000       0.050       0.059
==============================================================================
Omnibus:                        0.027   Durbin-Watson:                   2.196
Prob(Omnibus):                  0.987   Jarque-Bera (JB):                0.150
Skew:                          -0.006   Prob(JB):                        0.928
Kurtosis:                       2.840   Cond. No.                         328.
==============================================================================

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

Looking at some key statistics from the summary

The values we are concerned with are -

1. The coefficients and significance (p-values)

2. R-squared
3. F statistic and its significance

1. The coefficient for TV is 0.054, with a very low p value

The coefficient is statistically significant. So the association is not purely by chance.

2. R - squared is 0.816

Meaning that 81.6% of the variance in `Sales` is explained by `TV`

This is a decent R-squared value.

3. F statistic has a very low p value (practically low)

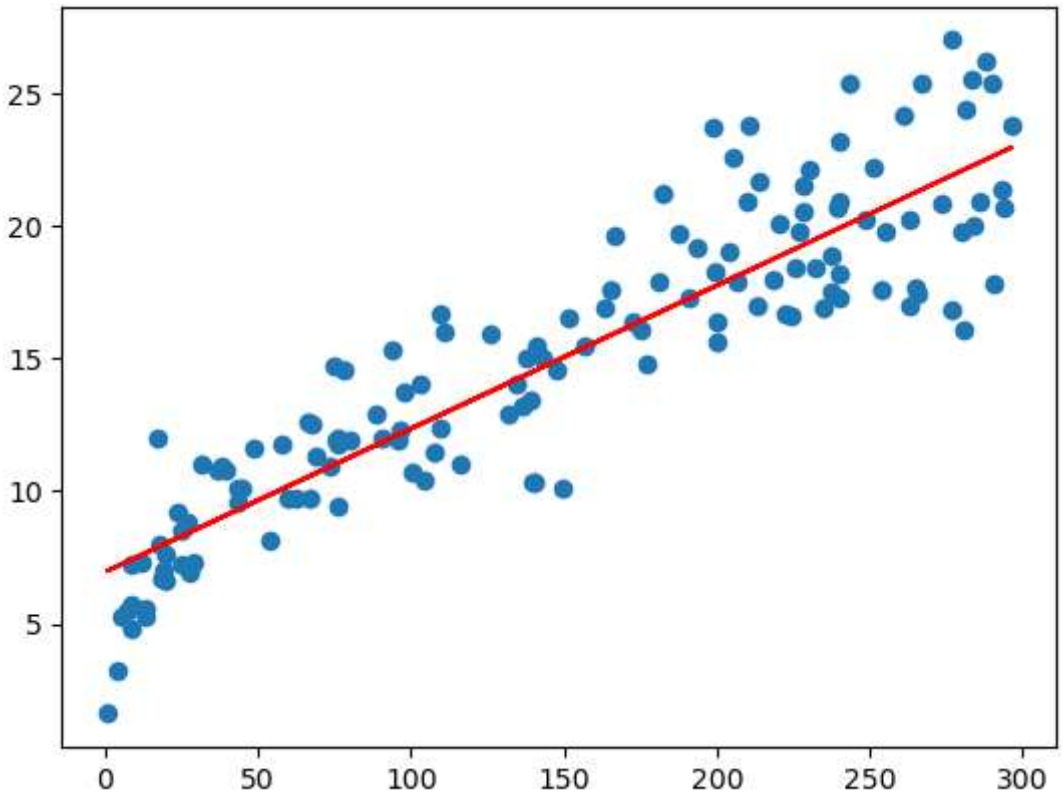
Meaning that the model fit is statistically significant, and the explained variance isn't purely by chance.

The fit is significant. Let's visualize how well the model fit the data.

From the parameters that we get, our linear regression equation becomes:

$$Sales = 6.948 + 0.054 \times TV$$

In [22]:
```
plt.scatter(X_train, y_train)
plt.plot(X_train, 6.948 + 0.054*X_train, 'r')
plt.show()
```



6. Model Evaluation

Residual analysis

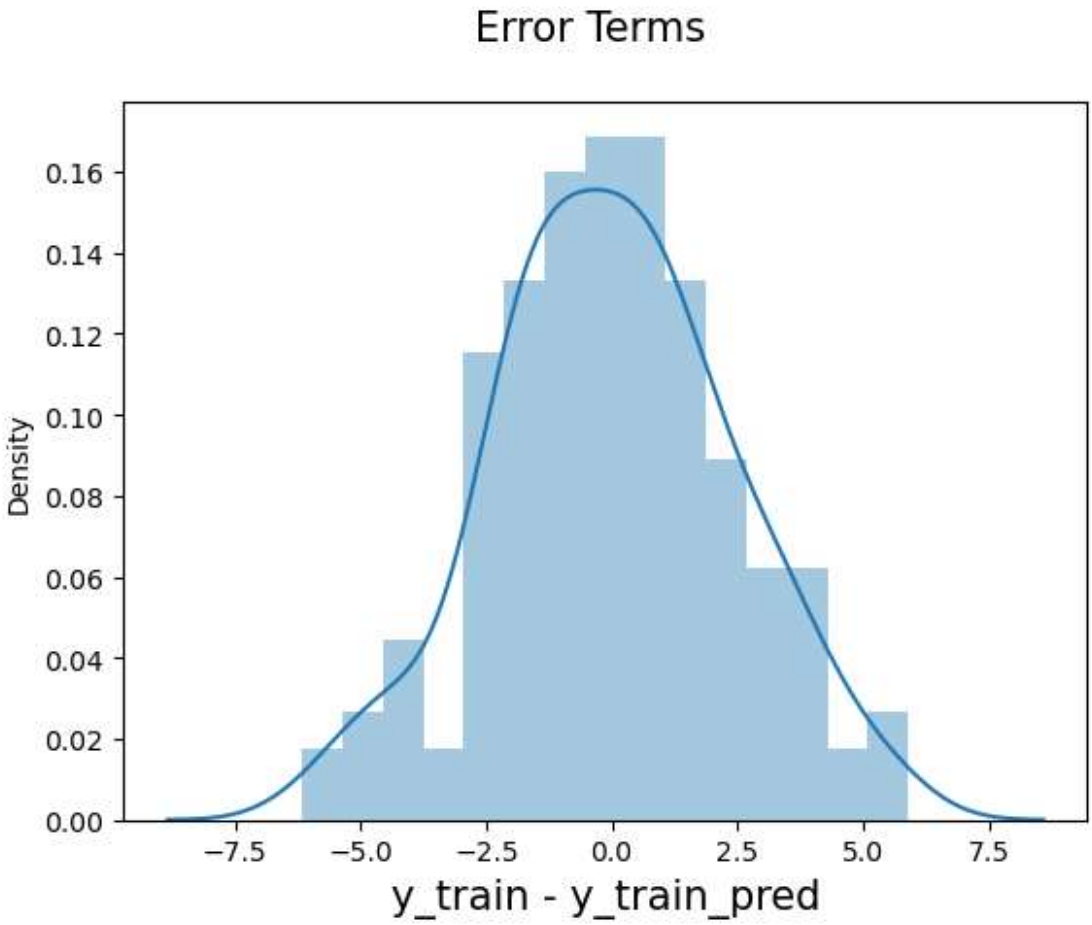
To validate assumptions of the model, and hence the reliability for inference

Distribution of the error terms

We need to check if the error terms are also normally distributed (which is infact, one of the major assumptions of linear regression), let us plot the histogram of the error terms and see what it looks like.

In [23]:
```
y_train_pred = lr.predict(X_train_sm)
res = (y_train - y_train_pred)
```

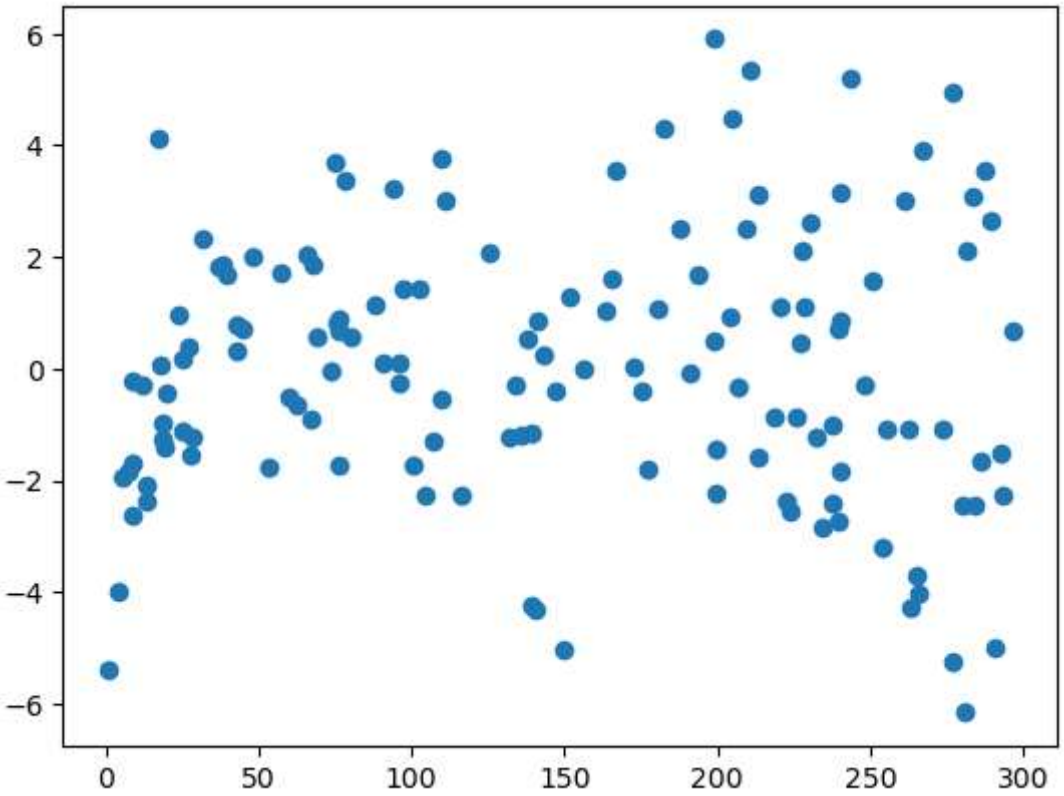
In [24]:
```
fig = plt.figure()
sns.distplot(res, bins = 15)
fig.suptitle('Error Terms', fontsize = 15)                  # Plot heading
plt.xlabel('y_train - y_train_pred', fontsize = 15)         # X-label
plt.show()
```



The residuals are following the normally distributed with a mean 0. All good!

Looking for patterns in the residuals

In [25]:
```
plt.scatter(X_train,res)
plt.show()
```



7. Predictions on the Test Set

Now that you have fitted a regression line on your train dataset, it's time to make some predictions on the test data. For this, you first need to add a constant to the `X_test` data like you did for `X_train` and then you can simply go on and predict the y values corresponding to `X_test` using the `predict` attribute of the fitted regression line.

In [26]:
```
# Add a constant to X_test
X_test_sm = sm.add_constant(X_test)

# Predict the y values corresponding to X_test_sm
y_pred = lr.predict(X_test_sm)
```

In [27]:
```
y_pred.head()
```

Out[27]:
```
126     7.374140
104    19.941482
99     14.323269
92     18.823294
111    20.132392
dtype: float64
```

In [28]:
```
from sklearn.metrics import mean_squared_error
from sklearn.metrics import r2_score
```

Looking at the RMSE

In [29]:
```
#Returns the mean squared error; we'll take a square root
np.sqrt(mean_squared_error(y_test, y_pred))
```

Out[29]: 2.019296008966233

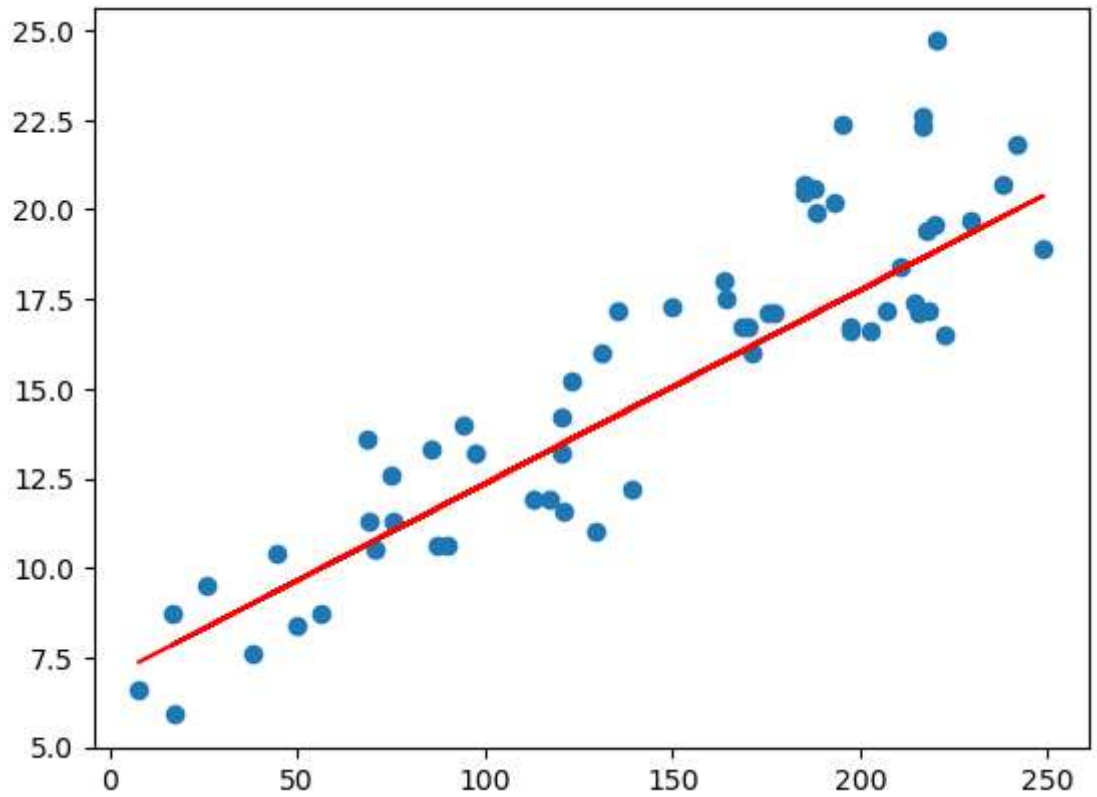
Checking the R-squared on the test set

In [30]:
```
r_squared = r2_score(y_test, y_pred)
r_squared
```

Out[30]: 0.7921031601245658

Visualizing the fit on the test set

In [31]:
```
plt.scatter(X_test, y_test)
plt.plot(X_test, 6.948 + 0.054 * X_test, 'r')
plt.show()
```



In [ ]: