Data

This sample data displays sales (in thousands of units) for a particular product as a function of advertising budgets (in thousands of dollars) for TV, radio, and newspaper media.

Indepenent variables

- TV: Advertising dollars spent on TV for a single product in a given market (in thousands of dollars)
- Radio: Advertising dollars spent on Radio
- Newspaper: Advertising dollars spent on Newspaper

Target Variable

- Sales: sales of a single product in a given market (in thousands of widgets)

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

In [2]:
```
# read data into a DataFrame
df = pd.read_csv("Advertising.csv")
df.head()
```

Out[2]:

	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	9.3
3	151.5	41.3	58.5	18.5
4	180.8	10.8	58.4	12.9

In [3]:
```
# print the shape of the DataFrame
df.shape
```

Out[3]:
```
(200, 4)
```

In [4]:
```
df.columns
```

Out[4]:
```
Index(['TV', 'radio', 'newspaper', 'sales'], dtype='object')
```

In [5]:
```
df.dtypes
```

Out[5]:
```
TV           float64
radio        float64
newspaper    float64
sales        float64
dtype: object
```

In [6]:
```
df.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 [7]:
```
df.isnull().sum()
```

Out[7]:
```
TV           0
radio        0
newspaper    0
sales        0
dtype: int64
```

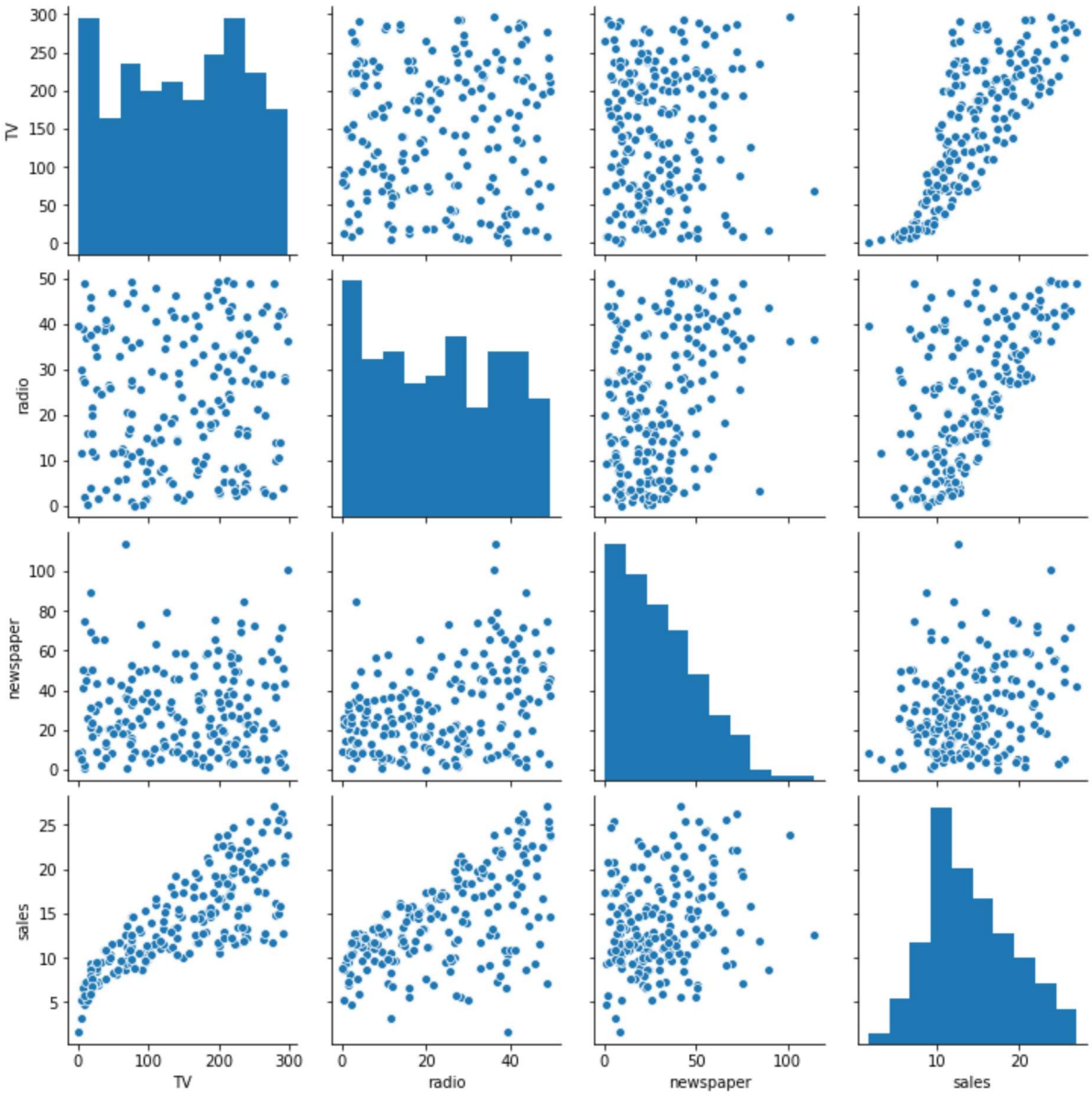
EDA

In [8]:
```
df.describe()
```

Out[8]:

	TV	radio	newspaper	sales
count	200.000000	200.000000	200.000000	200.000000
mean	147.042500	23.264000	30.554000	14.022500
std	85.854236	14.846809	21.778621	5.217457
min	0.700000	0.000000	0.300000	1.600000
25%	74.375000	9.975000	12.750000	10.375000
50%	149.750000	22.900000	25.750000	12.900000
75%	218.825000	36.525000	45.100000	17.400000
max	296.400000	49.600000	114.000000	27.000000

In [9]:
```
sns.pairplot(df)
plt.show()
```



In [10]:
```
df.corr()
```

Out[10]:

	TV	radio	newspaper	sales
TV	1.000000	0.054809	0.056648	0.782224
radio	0.054809	1.000000	0.354104	0.576223
newspaper	0.056648	0.354104	1.000000	0.228299
sales	0.782224	0.576223	0.228299	1.000000

In [11]:
```
# Everything BUT the sales column
X = df.drop('sales',axis=1)
y = df['sales']
```

Polynomial Regression with SciKit-Learn

We saw how to create a very simple best fit line, but now let's greatly expand our toolkit to start thinking about the considerations of overfitting, underfitting, model evaluation, as well as multiple features!

From Preprocessing, import PolynomialFeatures, which will help us transform our original data set by adding polynomial features

We will go from the equation in the form (shown here as if we only had one x feature):

$$\hat{y} = \beta_0 + \beta_1 x_1 + \epsilon$$

and create more features from the original x feature for some *d* degree of polynomial.

$$\hat{y} = \beta_0 + \beta_1 x_1 + \beta_1 x_1^2 + \ldots + \beta_d x_1^d + \epsilon$$

Then we can call the linear regression model on it, since in reality, we're just treating these new polynomial features x^2, x^3, ... x^d as new features. Obviously we need to be careful about choosing the correct value of *d* , the degree of the model. Our metric results on the test set will help us with this!

The other thing to note here is we have multiple X features, not just a single one as in the formula above, so in reality, the PolynomialFeatures will also take *interaction* terms into account for example, if an input sample is two dimensional and of the form [a, b], the degree-2 polynomial features are [1, a, b, a^2, ab, b^2].

In [12]:
```
from sklearn.preprocessing import PolynomialFeatures

polynomial_converter = PolynomialFeatures(degree=2,include_bias=False)
```

In [13]:
```
# polynomial_converter.fit(X)  # Converter "fits" to data, in this case, reads in every X column

# polynomial_converter.transform(X)  # Then it "transforms" and ouputs the new polynomial data
```

In [14]:
```
X_poly = polynomial_converter.fit_transform(X)
```

In [15]:
```
X_poly.shape
```

Out[15]:
```
(200, 9)
```

Train | Test Split

In [16]:
```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X_poly, y, test_size=0.3, random_state=101)
```

Model fitting on Polynomial Data

In [17]:
```
from sklearn.linear_model import LinearRegression

model = LinearRegression()

model.fit(X_train,y_train)
```

Out[17]:
```
LinearRegression()
```

Predictions

In [18]:
```
train_predictions = model.predict(X_train)
test_predictions = model.predict(X_test)
```

Evaluation

In [19]:
```
train_res = y_train - train_predictions
test_res = y_test - test_predictions
```

In [20]:
```
model.score(X_train,y_train)  # Train R2
```

Out[20]:
```
0.9868638137712757
```

In [21]:
```
model.score(X_test,y_test)   # Test R2
```

Out[21]:
```
0.9843529333146783
```

In [22]:
```
from sklearn.model_selection import cross_val_score
scores = cross_val_score(model,X_poly,y,cv=5)
scores

# Average of the MSE scores (we set back to positive)
abs(scores.mean())
```

Out[22]:
```
0.9842540981580088
```

In [23]:
```
from sklearn.metrics import mean_absolute_error,mean_squared_error
```

In [24]:
```
MAE = mean_absolute_error(y_test,test_predictions)
MAE
```

Out[24]:
```
0.4896798044803816
```

In [25]:
```
MSE = mean_squared_error(y_test,test_predictions)
MSE
```

Out[25]: 0.4417505510403749

In [26]:
```
RMSE = np.sqrt(MSE)
RMSE
```

Out[26]: 0.6646431757269271

Comparison with Linear Regression

Results on the Test Set (Note: Use the same Random Split to fairly compare!)

- Multiple Linear Regression:
  - MAE: 1.213
  - RMSE: 1.516
- Polynomial 2-degree:
  - MAE: 0.4896
  - RMSE: 0.664

Choosing a Model

Bias - Variance Tradeoff

Adjusting Parameters

Are we satisfied with this performance? Perhaps a higher order would improve performance even more! But how high is too high? It is now up to us to possibly go back and adjust our model and parameters, let's explore higher order Polynomials in a loop and plot out their error. This will nicely lead us into a discussion on Overfitting.

Let's use a for loop to do the following:

1. Create different order polynomial X data
2. Split that polynomial data for train/test
3. Fit on the training data
4. Report back the metrics on *both* the train and test results
5. Plot these results and explore overfitting

In [27]:
```
# TRAINING ERROR PER DEGREE
train_rmse_errors = []

# TEST ERROR PER DEGREE
test_rmse_errors = []

for d in range(1,10):

    # CREATE POLY DATA SET FOR DEGREE "d"
    polynomial_converter = PolynomialFeatures(degree=d,include_bias=False)
    X_poly = polynomial_converter.fit_transform(X)

    # SPLIT THIS NEW POLY DATA SET
    X_train, X_test, y_train, y_test = train_test_split(X_poly, y, test_size=
0.3, random_state=101)

    # TRAIN ON THIS NEW POLY SET
    model = LinearRegression()
    model.fit(X_train,y_train)

    # PREDICT ON BOTH TRAIN AND TEST
    train_pred = model.predict(X_train)
    test_pred = model.predict(X_test)

    # Calculate Errors

    # Errors on Train Set
    train_RMSE = np.sqrt(mean_squared_error(y_train,train_pred))

    # Errors on Test Set
    test_RMSE = np.sqrt(mean_squared_error(y_test,test_pred))

    # Append errors to lists for plotting later
    train_rmse_errors.append(train_RMSE)
    test_rmse_errors.append(test_RMSE)
```

In [28]:
```
train_rmse_errors
```

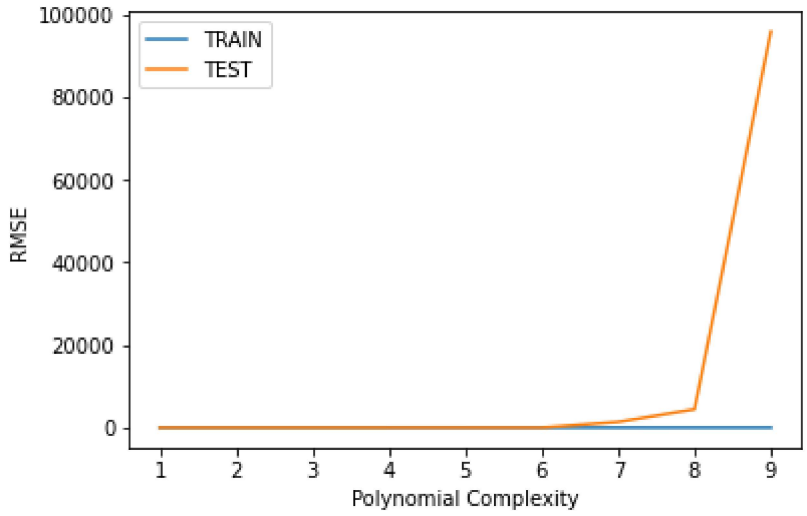
Out[28]:
```
[1.7345941243293759,
 0.5879574085292228,
 0.4339344356902065,
 0.351708368839935,
 0.250934294703179,
 0.19704459846551498,
 5.421420485986767,
 0.1418059854756517,
 0.16654227321858625]
```

In [29]:
```
test_rmse_errors
```

Out[29]:
```
[1.5161519375993877,
 0.6646431757269271,
 0.5803286825165038,
 0.5077742649213964,
 2.5758311664662017,
 4.4926997025114845,
 1381.4044216899795,
 4449.599748615518,
 95891.24543526048]
```

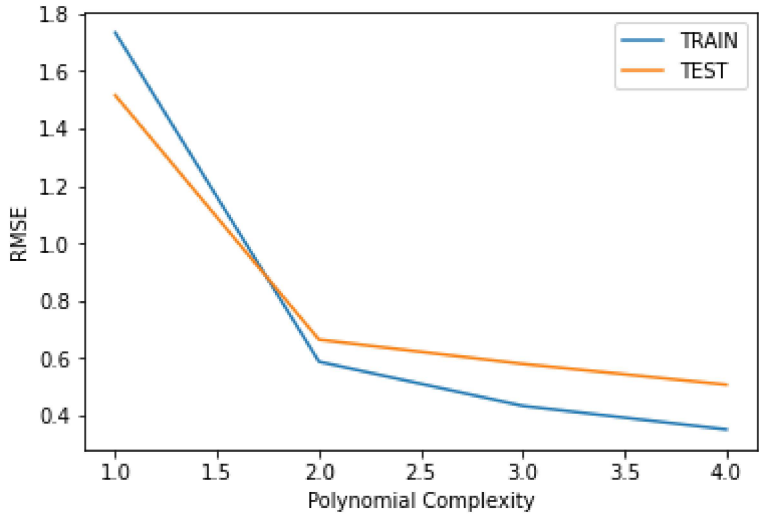
In [30]:
```
plt.plot(range(1,10),train_rmse_errors,label='TRAIN')
plt.plot(range(1,10),test_rmse_errors,label='TEST')
plt.xlabel("Polynomial Complexity")
plt.ylabel("RMSE")
plt.legend()
```

Out[30]:
```
<matplotlib.legend.Legend at 0x255b75f5880>
```



In [31]:
```
plt.plot(range(1,5),train_rmse_errors[:4],label='TRAIN')
plt.plot(range(1,5),test_rmse_errors[:4],label='TEST')
plt.xlabel("Polynomial Complexity")
plt.ylabel("RMSE")
plt.legend()
```

Out[31]:
```
<matplotlib.legend.Legend at 0x255b871b670>
```



Finalizing Model Choice

There are now 2 things we need to save, the Polynomial Feature creator AND the model itself. Let's explore how we would proceed from here:

1. Choose final parameters based on test metrics
2. Retrain on all data
3. Save Polynomial Converter object
4. Save model

In [32]:
```
# Based on our chart, could have also been degree=4, but
# it is better to be on the safe side of complexity
final_poly_converter = PolynomialFeatures(degree=3,include_bias=False)
```

In [33]:
```
final_model = LinearRegression()
```

In [34]:
```
final_model.fit(final_poly_converter.fit_transform(X),y)
```

Out[34]:
```
LinearRegression()
```

Saving Model and Converter

In [35]:
```
from joblib import dump, load
```

In [36]:
```
dump(final_model, 'sales_poly_model.joblib')
```

Out[36]:
```
['sales_poly_model.joblib']
```

In [37]:
```
dump(final_poly_converter,'poly_converter.joblib')
```

Out[37]:
```
['poly_converter.joblib']
```

Deployment and Predictions

Prediction on New Data

Recall that we will need to **convert** any incoming data to polynomial data, since that is what our model is trained on. We simply load up our saved converter object and only call **.transform()** on the new data, since we're not refitting to a new data set.

Our next ad campaign will have a total spend of 149k on TV, 22k on Radio, and 12k on Newspaper Ads, how many units could we expect to sell as a result of this?

In [38]:
```
loaded_poly = load('poly_converter.joblib')
loaded_model = load('sales_poly_model.joblib')
```

In [39]:
```
campaign_poly = loaded_poly.transform([[149,22,12]])
```

In [40]:
```
final_model.predict(campaign_poly)
```

Out[40]:
```
array([14.64501014])
```