Introduction to Simple Linear Regression

In this very simple example, we'll explore how to create a very simple fit line, the classic case of y=mx+b. We'll go carefully through each step, so you can see what type of question a simple fit line can answer. Keep in mind, this case is very simplified and is not the approach we'll take later on, its just here to get you thinking about linear regression in perhaps the same way Galton did.

Imports

```
import numpy as np
import pandas as pd
from matplotlib import pyplot as plt
import seaborn as sns
```

Sample Data

This sample data is from ISLR. It 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.

```
# Loading data
df=pd.read_csv("D:\\Study\\Programming\\python\\Python course from
udemy\\Udemy - 2022 Python for Machine Learning & Data Science
Masterclass\\01 - Introduction to Course\\1UNZIP-FOR-NOTEBOOKS-FINAL\\
08-Linear-Regression-Models\\Advertising.csv")
df.head()
```

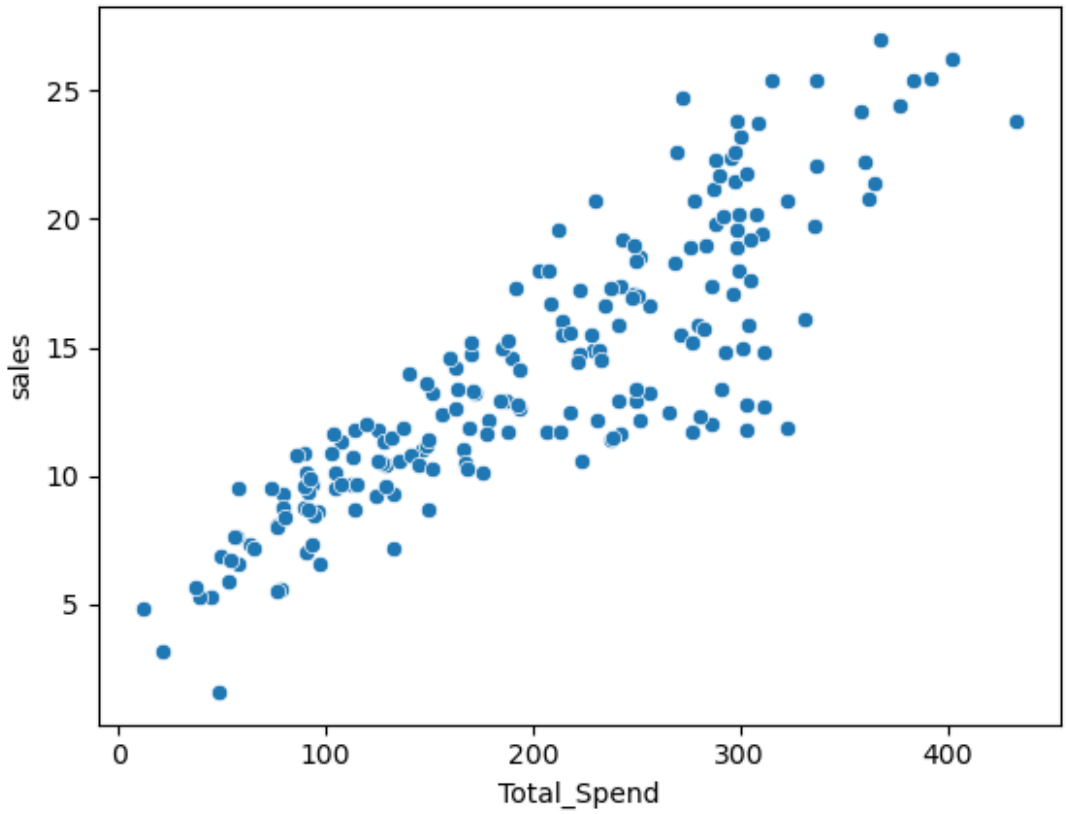
```
      TV  radio  newspaper  sales
0  230.1   37.8       69.2   22.1
1   44.5   39.3       45.1   10.4
2   17.2   45.9       69.3    9.3
3  151.5   41.3       58.5   18.5
4  180.8   10.8       58.4   12.9
```

Is there a relationship between *total* advertising spend and *sales*?

```
df['Total_Spend'] = df['TV'] + df['radio'] + df['newspaper']
df.head()
```

```
      TV  radio  newspaper  sales  Total_Spend
0  230.1   37.8       69.2   22.1        337.1
1   44.5   39.3       45.1   10.4        128.9
2   17.2   45.9       69.3    9.3        132.4
3  151.5   41.3       58.5   18.5        251.3
4  180.8   10.8       58.4   12.9        250.0
```

```
# Normal Scatterplot
sns.scatterplot(x='Total_Spend',y='sales',data=df);
```



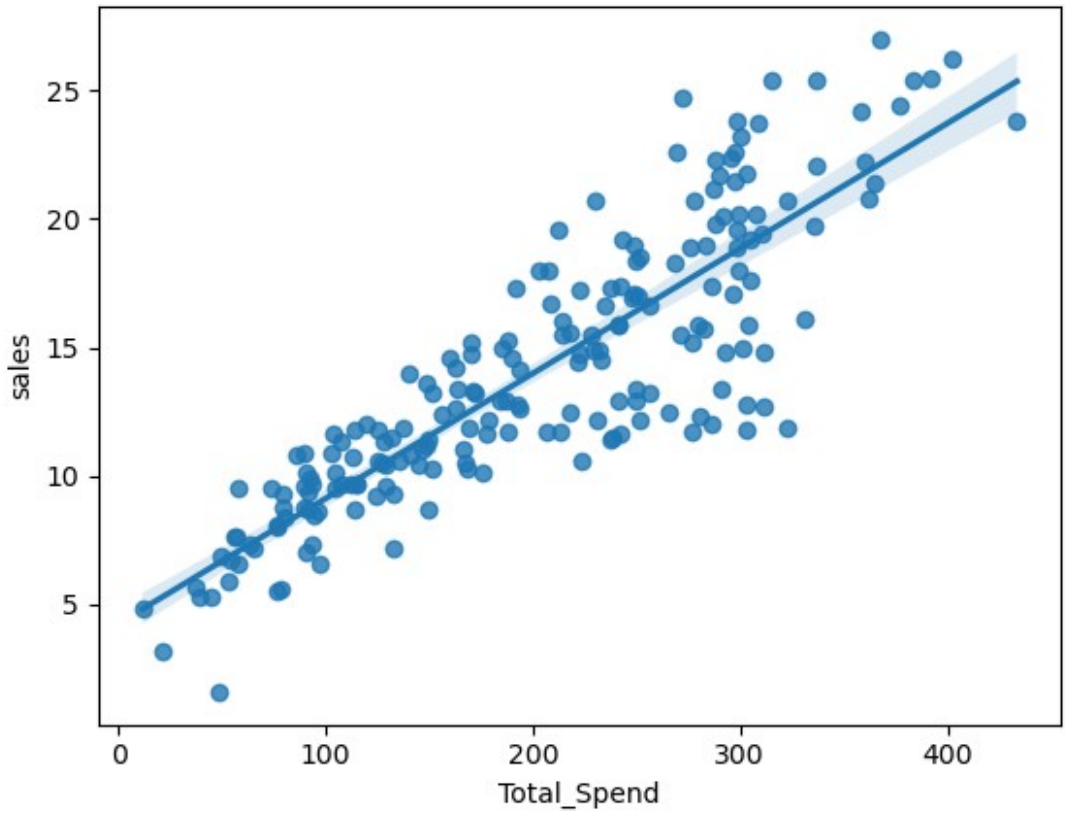
Least Squares Line

Full formulas available on Wikipedia: https://en.wikipedia.org/wiki/Linear_regression ,as well as in ISLR reading.

Understanding what a line of best fit answers. If someone was to spend a total of $200 , what would the expected sales be? We have simplified this quite a bit by combining all the features into "total spend", but we will come back to individual features later on. For now, let's focus on understanding what a linear regression line can help answer.

Our next ad campaign will have a total spend of $200, how many units do we expect to sell as a result of this?

```
# here we use regplot that show regression line(Best Fit Line) with
scatterplot
# Basically, we want to figure out how to create this line
sns.regplot(x='Total_Spend',y='sales',data=df);
```



Let's go ahead and start solving:

$$y = mx + b$$

Simply solve for m and b, remember, that as shown in the video, we are solving in a generalized form:

$$\hat{y} = \beta_0 + \beta_1 X$$

Capitalized to signal that we are dealing with a matrix of values, we have a known matrix of labels (sales numbers) Y and a known matrix of total_spend (X). We are going to solve for the *beta* coefficients, which as we expand to more than just a single feature, will be important to build an understanding of what features have the most predictive power. We use y hat to indicate that y hat is a prediction or estimation, y would be a true label/known value.

We can use NumPy for this (if you really wanted to, you could solve this by hand)

```
X= df['Total_Spend']
Y= df['sales']

# y=mx+b
# y=B1x + B0
help(np.polyfit)
```

```
Help on function polyfit in module numpy:

polyfit(x, y, deg, rcond=None, full=False, w=None, cov=False)
    Least squares polynomial fit.

    .. note::
       This forms part of the old polynomial API. Since version 1.4,
the
       new polynomial API defined in `numpy.polynomial` is preferred.
       A summary of the differences can be found in the
       :doc:`transition guide </reference/routines.polynomials>`.

    Fit a polynomial ``p(x) = p[0] * x**deg + ... + p[deg]`` of degree
`deg`
    to points `(x, y)`. Returns a vector of coefficients `p` that
minimises
    the squared error in the order `deg`, `deg-1`, ... `0`.

    The `Polynomial.fit <numpy.polynomial.polynomial.Polynomial.fit>`
class
    method is recommended for new code as it is more stable
numerically. See
    the documentation of the method for more information.

    Parameters
    ----------
    x : array_like, shape (M,)
        x-coordinates of the M sample points ``(x[i], y[i])``.
    y : array_like, shape (M,) or (M, K)
        y-coordinates of the sample points. Several data sets of
sample
        points sharing the same x-coordinates can be fitted at once by
        passing in a 2D-array that contains one dataset per column.
    deg : int
        Degree of the fitting polynomial
    rcond : float, optional
        Relative condition number of the fit. Singular values smaller
than
        this relative to the largest singular value will be ignored.
The
        default value is len(x)*eps, where eps is the relative
precision of
        the float type, about 2e-16 in most cases.
    full : bool, optional
        Switch determining nature of return value. When it is False
(the
        default) just the coefficients are returned, when True
diagnostic
        information from the singular value decomposition is also
returned.
```

```
    w : array_like, shape (M,), optional
        Weights to apply to the y-coordinates of the sample points.
For
        gaussian uncertainties, use 1/sigma (not 1/sigma**2).
    cov : bool or str, optional
        If given and not `False`, return not just the estimate but
also its
        covariance matrix. By default, the covariance are scaled by
        chi2/dof, where dof = M - (deg + 1), i.e., the weights are
presumed
        to be unreliable except in a relative sense and everything is
scaled
        such that the reduced chi2 is unity. This scaling is omitted
if
        ``cov='unscaled'``, as is relevant for the case that the
weights are
        1/sigma**2, with sigma known to be a reliable estimate of the
        uncertainty.

    Returns
    -------
    p : ndarray, shape (deg + 1,) or (deg + 1, K)
        Polynomial coefficients, highest power first.  If `y` was 2-D,
the
        coefficients for `k`-th data set are in ``p[:,k]``.

    residuals, rank, singular_values, rcond
        Present only if `full` = True.  Residuals is sum of squared
residuals
        of the least-squares fit, the effective rank of the scaled
Vandermonde
        coefficient matrix, its singular values, and the specified
value of
        `rcond`. For more details, see `linalg.lstsq`.

    V : ndarray, shape (M,M) or (M,M,K)
        Present only if `full` = False and `cov`=True.  The covariance
        matrix of the polynomial coefficient estimates.  The diagonal
of
        this matrix are the variance estimates for each coefficient.
If y
        is a 2-D array, then the covariance matrix for the `k`-th data
set
        are in ``V[:,:,k]``


    Warns
    -----
    RankWarning
        The rank of the coefficient matrix in the least-squares fit is
```

```
        deficient. The warning is only raised if `full` = False.

        The warnings can be turned off by

        >>> import warnings
        >>> warnings.simplefilter('ignore', np.RankWarning)

    See Also
    --------
    polyval : Compute polynomial values.
    linalg.lstsq : Computes a least-squares fit.
    scipy.interpolate.UnivariateSpline : Computes spline fits.

    Notes
    -----
    The solution minimizes the squared error

    .. math ::
        E = \sum_{j=0}^k |p(x_j) - y_j|^2

    in the equations::

        x[0]**n * p[0] + ... + x[0] * p[n-1] + p[n] = y[0]
        x[1]**n * p[0] + ... + x[1] * p[n-1] + p[n] = y[1]
        ...
        x[k]**n * p[0] + ... + x[k] * p[n-1] + p[n] = y[k]

    The coefficient matrix of the coefficients `p` is a Vandermonde
matrix.

    `polyfit` issues a `RankWarning` when the least-squares fit is
badly
    conditioned. This implies that the best fit is not well-defined
due
    to numerical error. The results may be improved by lowering the
polynomial
    degree or by replacing `x` by `x` - `x`.mean(). The `rcond`
parameter
    can also be set to a value smaller than its default, but the
resulting
    fit may be spurious: including contributions from the small
singular
    values can add numerical noise to the result.

    Note that fitting polynomial coefficients is inherently badly
conditioned
    when the degree of the polynomial is large or the interval of
sample points
    is badly centered. The quality of the fit should always be checked
in these
```

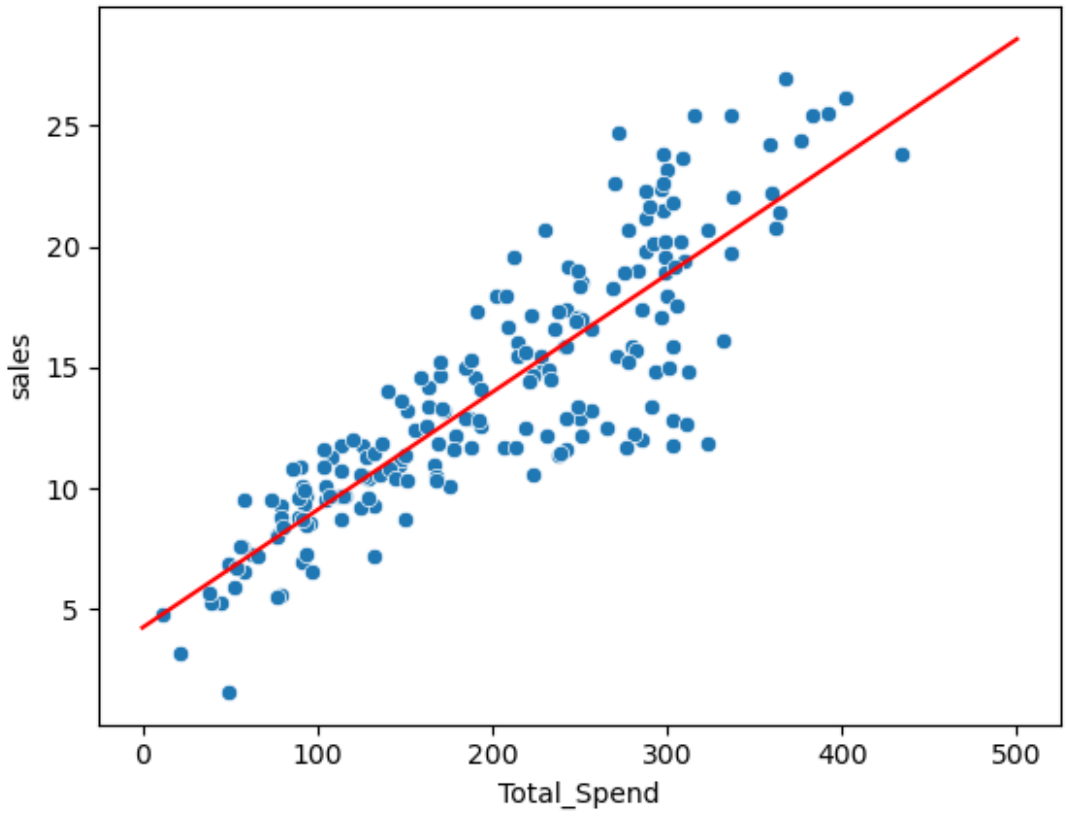
```
    cases. When polynomial fits are not satisfactory, splines may be a
good
    alternative.

    References
    ----------
    .. [1] Wikipedia, "Curve fitting",
           https://en.wikipedia.org/wiki/Curve_fitting
    .. [2] Wikipedia, "Polynomial interpolation",
           https://en.wikipedia.org/wiki/Polynomial_interpolation

    Examples
    --------
    >>> import warnings
    >>> x = np.array([0.0, 1.0, 2.0, 3.0,  4.0,  5.0])
    >>> y = np.array([0.0, 0.8, 0.9, 0.1, -0.8, -1.0])
    >>> z = np.polyfit(x, y, 3)
    >>> z
    array([ 0.08703704, -0.81349206,  1.69312169, -0.03968254]) # may
vary

    It is convenient to use `poly1d` objects for dealing with
polynomials:

    >>> p = np.poly1d(z)
    >>> p(0.5)
    0.6143849206349179 # may vary
    >>> p(3.5)
    -0.34732142857143039 # may vary
    >>> p(10)
    22.579365079365115 # may vary

    High-order polynomials may oscillate wildly:

    >>> with warnings.catch_warnings():
    ...     warnings.simplefilter('ignore', np.RankWarning)
    ...     p30 = np.poly1d(np.polyfit(x, y, 30))
    ...
    >>> p30(4)
    -0.80000000000000204 # may vary
    >>> p30(5)
    -0.99999999999999445 # may vary
    >>> p30(4.5)
    -0.10547061179440398 # may vary

    Illustration:

    >>> import matplotlib.pyplot as plt
    >>> xp = np.linspace(-2, 6, 100)
    >>> _ = plt.plot(x, y, '.', xp, p(xp), '-', xp, p30(xp), '--')
```

```
>>> plt.ylim(-2,2)
(-2, 2)
>>> plt.show()
```

```
# These are values of B1(m), B0(b)
np.polyfit(X,Y,deg=1) # Returns highest order coef first!
```

```
array([0.04868788, 4.24302822])
```

```
# Here we created 100 linear spaced
# Potential Future Spend Budgets
potential_spend=np.linspace(0,500,100)

# y= mx+b Here we are creating
predicted_sales=0.04868788*potential_spend+4.24302822

# Here this is the same regresion line which we have plot by regplot ,
this is step by step method of that
# This is liner fit
sns.scatterplot(x='Total_Spend',y='sales',data=df)
plt.plot(potential_spend,predicted_sales,color='red');
```



Our next ad campaign will have a total spend of $200, how many units do we expect to sell as a result of this?

```
# Now we can calculate our spend and calculate back predicted sales
spend=200

predicted_sales=0.04868788*200 + 4.24302822
predicted_sales
```

```
13.98060422
```

Further considerations...which we will explore in much more depth!

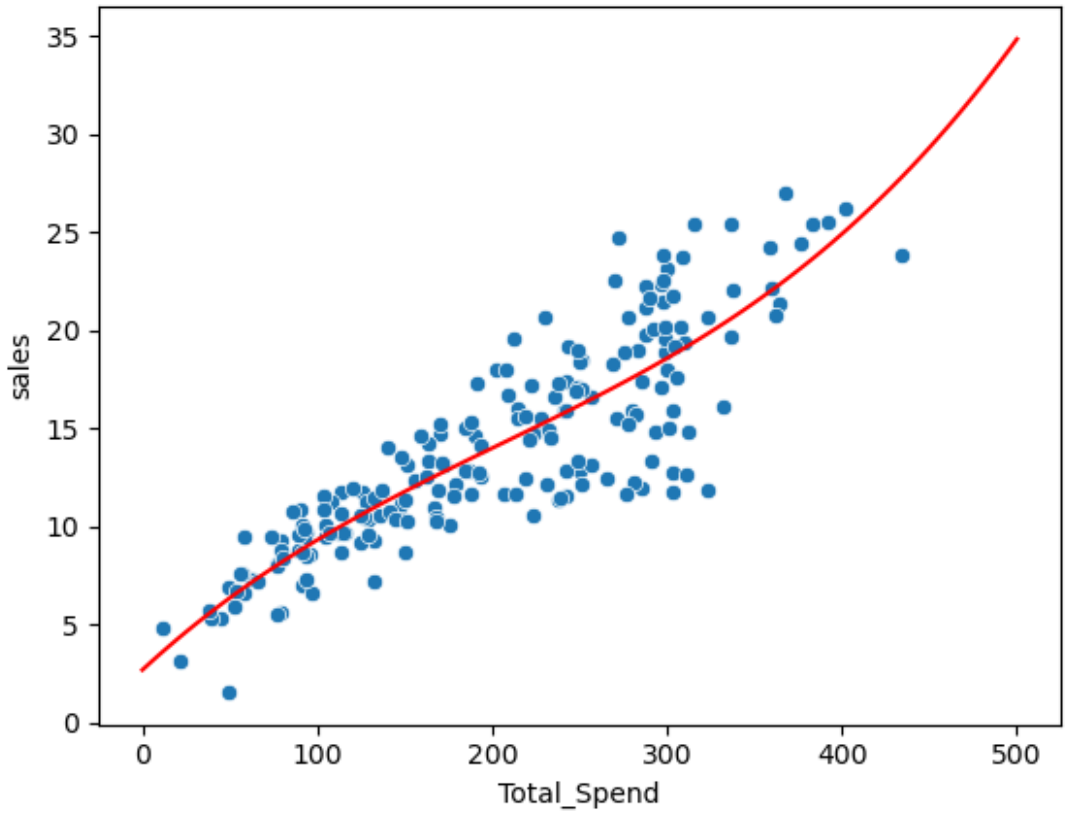
Overfitting, Underfitting, and Measuring Performance

Notice we fit to order=1 , essentially a straight line, we can begin to explore higher orders, but does higher order mean an overall better fit? Is it possible to fit too much? Too little? How would we know and how do we even define a good fit?

```
# y=B1x + B0

#y= B3x**3 + B2x**2 + B1x + B0 ( Fo rdegree = 3)
np.polyfit(X,Y,3) # But here we can see the x**3 is very samll so we
can ignore that too same as
```

```
array([ 3.07615033e-07, -1.89392449e-04,  8.20886302e-02,
2.70495053e+00])
```

```
pot_sales= 3.07615033e-07*potential_spend**3 + -1.89392449e-
04*potential_spend**2+  8.20886302e-02*potential_spend +
2.70495053e+00

# This Curve Fit
sns.scatterplot(x='Total_Spend',y='sales',data=df)
plt.plot(potential_spend,pot_sales,color='red');
```



Is this better than our straight line fit? What are good ways of measuring this?

Multiple Features

The real data had 3 features, not everything in total spend, this would allow us to repeat the process and maybe get a more accurate result?

```
X = df[['TV','radio','newspaper']]
Y = df['sales']
```

```
# Note here we're passing in 3 which matches up with 3 unique
features, so we're not polynomial yet
np.polyfit(X,Y,1)
```

```
---------------------------------------------------------------------------
-----
TypeError                                 Traceback (most recent call
last)
~\AppData\Local\Temp\ipykernel_6588\4146141824.py in <module>
      1 # Note here we're passing in 3 which matches up with 3 unique
features, so we're not polynomial yet
----> 2 np.polyfit(X,Y,1)

<__array_function__ internals> in polyfit(*args, **kwargs)
```

```
C:\ProgramData\Anaconda3\lib\site-packages\numpy\lib\polynomial.py in
polyfit(x, y, deg, rcond, full, w, cov)
    626         raise ValueError("expected deg >= 0")
    627     if x.ndim != 1:
--> 628         raise TypeError("expected 1D vector for x")
    629     if x.size == 0:
    630         raise TypeError("expected non-empty vector for x")

TypeError: expected 1D vector for x
```

Uh oh! Polyfit only works with a 1D X array! We'll need to move on to a more powerful library...

Linear 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!

```
df.head()
```

```
      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
```

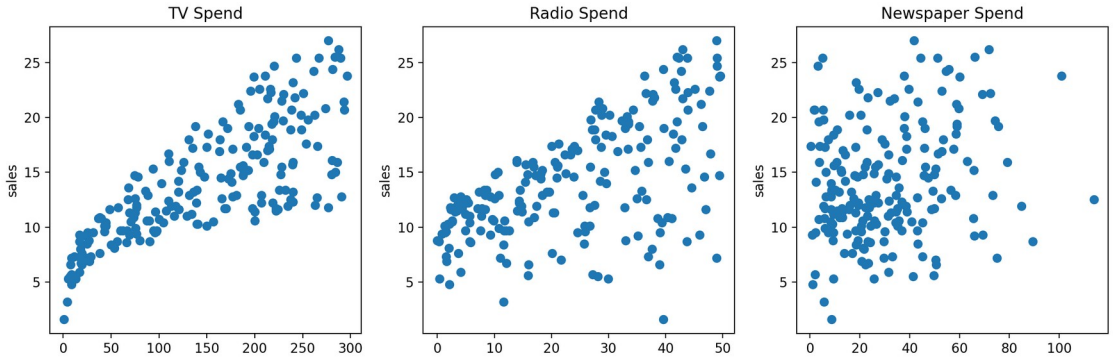
Expanding the Questions

Previously, we explored **Is there a relationship between *total* advertising spend and *sales*?** as well as predicting the total sales for some value of total spend. Now we want to expand this to **What is the relationship between each advertising channel (TV,Radio,Newspaper) and sales?**

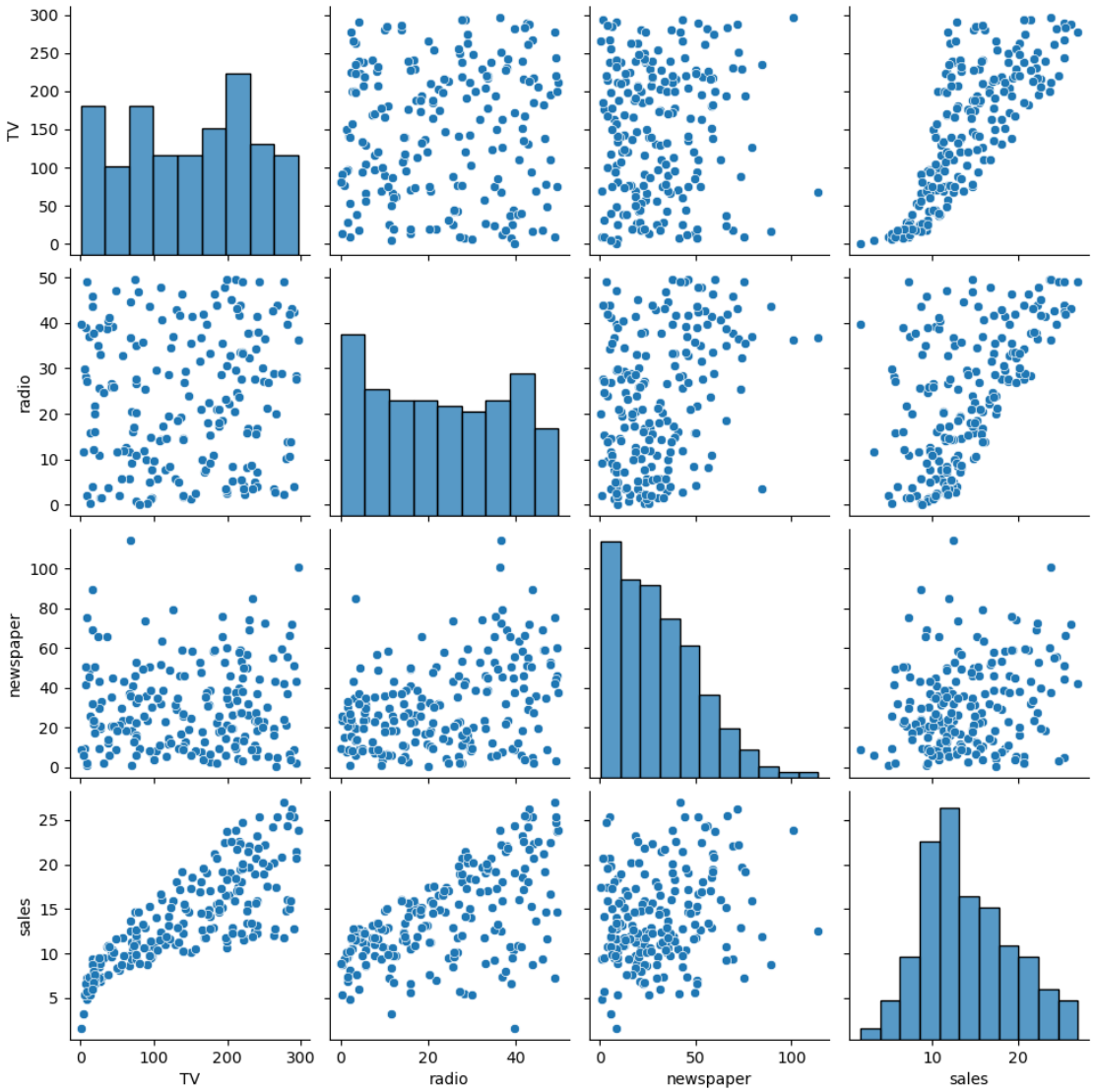
Multiple Features (N-Dimensional)

```
fig, axes = plt.subplots(nrows=1, ncols=3,figsize=(12,4),dpi=200)

axes[0].plot(df['TV'],df['sales'],'o') # Here 'o' is shape of markers
axes[0].set_ylabel("sales")
axes[0].set_title("TV Spend")

axes[1].plot(df['radio'],df['sales'],'o')
axes[1].set_ylabel("sales")
axes[1].set_title("Radio Spend")

axes[2].plot(df['newspaper'],df['sales'],'o')
```

```
axes[2].set_ylabel("sales")
axes[2].set_title("Newspaper Spend")

plt.tight_layout();
```




```
# Relationships between features
sns.pairplot(data=df);# we can choose diag_kind='kde' for kde plot at
diagonals
```



Introducing SciKit Learn

We will work a lot with the scitkit learn library, so get comfortable with its model estimator syntax, as well as exploring its incredibly useful documentation!

```
# For isntalling sklearn for first time
pip install -U scikit-learn
```

```
  File "C:\Users\Chromsy\AppData\Local\Temp\
ipykernel_6588\3184758135.py", line 2
    pip install -U scikit-learn
        ^
SyntaxError: invalid syntax
```

```
X = df.drop('sales',axis=1) #Here we prepare data that we work on
y = df['sales'] # we seprate sales in different DataFrame that we have
to predict
```

Train | Test Split

Make sure you have watched the Machine Learning Overview videos on Supervised Learning to understand why we do this step

```
#pip install -U scikit-learn

from sklearn.model_selection import train_test_split

# random_state:
# https://stackoverflow.com/questions/28064634/random-state-pseudo-
random-number-in-scikit-learn
X_train, X_test, Y_train, Y_test = train_test_split(X, Y,
test_size=0.3, random_state=101)
# Here test_size is percent of data we want i your test
# random_state is like seed in numpy, to fix random number on every
time we hit same seed number

# Here we see X_train
X_train
```

```
        TV  radio  newspaper
85   193.2   18.4       65.7
183  287.6   43.0       71.8
127   80.2    0.0        9.2
53   182.6   46.2       58.7
100  222.4    4.3       49.8
..     ...    ...        ...
63   102.7   29.6        8.4
70   199.1   30.6       38.7
81   239.8    4.1       36.9
11   214.7   24.0        4.0
95   163.3   31.6       52.9

[140 rows x 3 columns]
```

```
# Here we see Y_train
Y_train
```

```
85     15.2
183    26.2
127     8.8
53     21.2
100    11.7
       ...
63     14.0
70     18.3
81     12.3
```

```
11     17.4
95     16.9
Name: sales, Length: 140, dtype: float64
```

```
X_test[0:5]
```

```
        TV  radio  newspaper
37    74.7   49.4       45.7
109  255.4   26.9        5.5
31   112.9   17.4       38.6
89   109.8   47.8       51.4
66    31.5   24.6        2.2
```

```
Y_test[0:5]
```

```
37     14.7
109    19.8
31     11.9
89     16.7
66      9.5
Name: sales, dtype: float64
```

```
# To check the number of data in test and train as per 70 and 30%
len(df) , len(X_train),len(X_test)
```

```
(200, 140, 60)
```

Creating a Model (Estimator)

Import a model class from a model family

```
from sklearn.linear_model import LinearRegression
```

```
help(LinearRegression)
```

```
Help on class LinearRegression in module sklearn.linear_model._base:

class LinearRegression(sklearn.base.MultiOutputMixin,
sklearn.base.RegressorMixin, LinearModel)
 |  LinearRegression(*, fit_intercept=True, copy_X=True, n_jobs=None,
positive=False)
 |
 |  Ordinary least squares Linear Regression.
 |
 |  LinearRegression fits a linear model with coefficients w =
(w1, ..., wp)
 |  to minimize the residual sum of squares between the observed
targets in
 |  the dataset, and the targets predicted by the linear
approximation.
 |
 |  Parameters
 |  ----------
```

```
 |  fit_intercept : bool, default=True
 |      Whether to calculate the intercept for this model. If set
 |      to False, no intercept will be used in calculations
 |      (i.e. data is expected to be centered).
 |
 |  copy_X : bool, default=True
 |      If True, X will be copied; else, it may be overwritten.
 |
 |  n_jobs : int, default=None
 |      The number of jobs to use for the computation. This will only
provide
 |      speedup in case of sufficiently large problems, that is if
firstly
 |      `n_targets > 1` and secondly `X` is sparse or if `positive` is
set
 |      to `True`. ``None`` means 1 unless in a
 |      :obj:`joblib.parallel_backend` context. ``-1`` means using all
 |      processors. See :term:`Glossary <n_jobs>` for more details.
 |
 |  positive : bool, default=False
 |      When set to ``True``, forces the coefficients to be positive.
This
 |      option is only supported for dense arrays.
 |
 |      .. versionadded:: 0.24
 |
 |  Attributes
 |  ----------
 |  coef_ : array of shape (n_features, ) or (n_targets, n_features)
 |      Estimated coefficients for the linear regression problem.
 |      If multiple targets are passed during the fit (y 2D), this
 |      is a 2D array of shape (n_targets, n_features), while if only
 |      one target is passed, this is a 1D array of length n_features.
 |
 |  rank_ : int
 |      Rank of matrix `X`. Only available when `X` is dense.
 |
 |  singular_ : array of shape (min(X, y),)
 |      Singular values of `X`. Only available when `X` is dense.
 |
 |  intercept_ : float or array of shape (n_targets,)
 |      Independent term in the linear model. Set to 0.0 if
 |      `fit_intercept = False`.
 |
 |  n_features_in_ : int
 |      Number of features seen during :term:`fit`.
 |
 |      .. versionadded:: 0.24
 |
 |  feature_names_in_ : ndarray of shape (`n_features_in_`,)
```

```
 |      Names of features seen during :term:`fit`. Defined only when
`X`
 |      has feature names that are all strings.
 |
 |      .. versionadded:: 1.0
 |
 |  See Also
 |  --------
 |  Ridge : Ridge regression addresses some of the
 |      problems of Ordinary Least Squares by imposing a penalty on
the
 |      size of the coefficients with l2 regularization.
 |  Lasso : The Lasso is a linear model that estimates
 |      sparse coefficients with l1 regularization.
 |  ElasticNet : Elastic-Net is a linear regression
 |      model trained with both l1 and l2 -norm regularization of the
 |      coefficients.
 |
 |  Notes
 |  -----
 |  From the implementation point of view, this is just plain Ordinary
 |  Least Squares (scipy.linalg.lstsq) or Non Negative Least Squares
 |  (scipy.optimize.nnls) wrapped as a predictor object.
 |
 |  Examples
 |  --------
 |  >>> import numpy as np
 |  >>> from sklearn.linear_model import LinearRegression
 |  >>> X = np.array([[1, 1], [1, 2], [2, 2], [2, 3]])
 |  >>> # y = 1 * x_0 + 2 * x_1 + 3
 |  >>> y = np.dot(X, np.array([1, 2])) + 3
 |  >>> reg = LinearRegression().fit(X, y)
 |  >>> reg.score(X, y)
 |  1.0
 |  >>> reg.coef_
 |  array([1., 2.])
 |  >>> reg.intercept_
 |  3.0...
 |  >>> reg.predict(np.array([[3, 5]]))
 |  array([16.])
 |
 |  Method resolution order:
 |      LinearRegression
 |      sklearn.base.MultiOutputMixin
 |      sklearn.base.RegressorMixin
 |      LinearModel
 |      sklearn.base.BaseEstimator
 |      builtins.object
 |
 |  Methods defined here:
```

```
 |
 |  __init__(self, *, fit_intercept=True, copy_X=True, n_jobs=None,
positive=False)
 |      Initialize self.  See help(type(self)) for accurate signature.
 |
 |  fit(self, X, y, sample_weight=None)
 |      Fit linear model.
 |
 |      Parameters
 |      ----------
 |      X : {array-like, sparse matrix} of shape (n_samples,
n_features)
 |          Training data.
 |
 |      y : array-like of shape (n_samples,) or (n_samples, n_targets)
 |          Target values. Will be cast to X's dtype if necessary.
 |
 |      sample_weight : array-like of shape (n_samples,), default=None
 |          Individual weights for each sample.
 |
 |          .. versionadded:: 0.17
 |             parameter *sample_weight* support to LinearRegression.
 |
 |      Returns
 |      -------
 |      self : object
 |          Fitted Estimator.
 |
 |
----------------------------------------------------------------------
 |  Data and other attributes defined here:
 |
 |  __abstractmethods__ = frozenset()
 |
 |  __annotations__ = {'_parameter_constraints': <class 'dict'>}
 |
 |
----------------------------------------------------------------------
 |  Data descriptors inherited from sklearn.base.MultiOutputMixin:
 |
 |  __dict__
 |      dictionary for instance variables (if defined)
 |
 |  __weakref__
 |      list of weak references to the object (if defined)
 |
 |
----------------------------------------------------------------------
 |  Methods inherited from sklearn.base.RegressorMixin:
 |
```

```
 |  score(self, X, y, sample_weight=None)
 |      Return the coefficient of determination of the prediction.
 |
 |      The coefficient of determination :math:`R^2` is defined as
 |      :math:`(1 - \frac{u}{v})`, where :math:`u` is the residual
 |      sum of squares ``((y_true - y_pred)** 2).sum()`` and :math:`v`
 |      is the total sum of squares ``((y_true - y_true.mean()) **
2).sum()``.
 |      The best possible score is 1.0 and it can be negative (because
the
 |      model can be arbitrarily worse). A constant model that always
predicts
 |      the expected value of `y`, disregarding the input features,
would get
 |      a :math:`R^2` score of 0.0.
 |
 |      Parameters
 |      ----------
 |      X : array-like of shape (n_samples, n_features)
 |          Test samples. For some estimators this may be a
precomputed
 |          kernel matrix or a list of generic objects instead with
shape
 |          ``(n_samples, n_samples_fitted)``, where
``n_samples_fitted``
 |          is the number of samples used in the fitting for the
estimator.
 |
 |      y : array-like of shape (n_samples,) or (n_samples, n_outputs)
 |          True values for `X`.
 |
 |      sample_weight : array-like of shape (n_samples,), default=None
 |          Sample weights.
 |
 |      Returns
 |      -------
 |      score : float
 |          :math:`R^2` of ``self.predict(X)`` wrt. `y`.
 |
 |      Notes
 |      -----
 |      The :math:`R^2` score used when calling ``score`` on a
regressor uses
 |      ``multioutput='uniform_average'`` from version 0.23 to keep
consistent
 |      with default value of :func:`~sklearn.metrics.r2_score`.
 |      This influences the ``score`` method of all the multioutput
 |      regressors (except for
 |      :class:`~sklearn.multioutput.MultiOutputRegressor`).
 |
```

```
 |
----------------------------------------------------------------------
 |  Methods inherited from LinearModel:
 |
 |  predict(self, X)
 |      Predict using the linear model.
 |
 |      Parameters
 |      ----------
 |      X : array-like or sparse matrix, shape (n_samples, n_features)
 |          Samples.
 |
 |      Returns
 |      -------
 |      C : array, shape (n_samples,)
 |          Returns predicted values.
 |
 |
----------------------------------------------------------------------
 |  Methods inherited from sklearn.base.BaseEstimator:
 |
 |  __getstate__(self)
 |
 |  __repr__(self, N_CHAR_MAX=700)
 |      Return repr(self).
 |
 |  __setstate__(self, state)
 |
 |  get_params(self, deep=True)
 |      Get parameters for this estimator.
 |
 |      Parameters
 |      ----------
 |      deep : bool, default=True
 |          If True, will return the parameters for this estimator and
 |          contained subobjects that are estimators.
 |
 |      Returns
 |      -------
 |      params : dict
 |          Parameter names mapped to their values.
 |
 |  set_params(self, **params)
 |      Set the parameters of this estimator.
 |
 |      The method works on simple estimators as well as on nested
objects
 |      (such as :class:`~sklearn.pipeline.Pipeline`). The latter have
 |      parameters of the form ``<component>__<parameter>`` so that
it's
```

```
|       possible to update each component of a nested object.
|
|       Parameters
|       ----------
|       **params : dict
|           Estimator parameters.
|
|       Returns
|       -------
|       self : estimator instance
|           Estimator instance.
```

```
model = LinearRegression()
```

```
model.fit(X_train,Y_train)
```

```
LinearRegression()
```

Understanding and utilizing the Model

Evaluation on the Test Set

Metrics

Make sure you've viewed the video on these metrics! The three most common evaluation metrics for regression problems:

Mean Absolute Error (MAE) is the mean of the absolute value of the errors:

$$\frac{1}{n}\sum_{i=1}^{n}\left|y_i-\hat{y}_i\right)$$

Mean Squared Error (MSE) is the mean of the squared errors:

$$\frac{1}{n}\sum_{i=1}^{n}\left(y_i-\hat{y}_i\right)^2$$

Root Mean Squared Error (RMSE) is the square root of the mean of the squared errors:

$$\sqrt{\frac{1}{n}\sum_{i=1}^{n}\left(y_i-\hat{y}_i\right)^2}$$

Comparing these metrics:

- **MAE** is the easiest to understand, because it's the average error.
- **MSE** is more popular than MAE, because MSE "punishes" larger errors, which tends to be useful in the real world.

- **RMSE** is even more popular than MSE, because RMSE is interpretable in the "y" units.

All of these are **loss functions**, because we want to minimize them.

Calculate Performance on Test Set

We want to fairly evaluate our model, so we get performance metrics on the test set (data the model has never seen before).

```
# We only pass in test features
# The model predicts its own y hat
# We can then compare these results to the true y test label value
test_predictions=model.predict(X_test)
test_predictions
```

```
array([15.74131332, 19.61062568, 11.44888935, 17.00819787,
9.17285676,
        7.01248287, 20.28992463, 17.29953992,  9.77584467,
19.22194224,
       12.40503154, 13.89234998, 13.72541098, 21.28794031,
18.42456638,
        9.98198406, 15.55228966,  7.68913693,  7.55614992,
20.40311209,
        7.79215204, 18.24214098, 24.68631904, 22.82199068,
7.97962085,
       12.65207264, 21.46925937,  8.05228573, 12.42315981,
12.50719678,
       10.77757812, 19.24460093, 10.070269  ,  6.70779999,
17.31492147,
        7.76764327,  9.25393336,  8.27834697, 10.58105585,
10.63591128,
       13.01002595,  9.77192057, 10.21469861,  8.04572042,
11.5671075 ,
       10.08368001,  8.99806574, 16.25388914, 13.23942315,
20.81493419,
       12.49727439, 13.96615898, 17.56285075, 11.14537013,
12.56261468,
        5.50870279, 23.29465134, 12.62409688, 18.77399978,
15.18785675])
```

```
from sklearn.metrics import mean_absolute_error,mean_squared_error
```

```
MAE=mean_absolute_error(Y_test,test_predictions)
MSE=mean_squared_error(Y_test,test_predictions)
RMSE= np.sqrt(MSE)
```

```
MAE,MSE,RMSE
```

```
(1.213745773614481, 2.2987166978863782, 1.516151937599388)
```

```
# Here we can see mean of data is 14 and error is approx 1.5 so we can
say that error is about 10%
df['sales'].mean()
```

```
14.022500000000003
```

Review our video to understand whether these values are "good enough". 010 Linear Regression - Residual Plots_en

Residuals

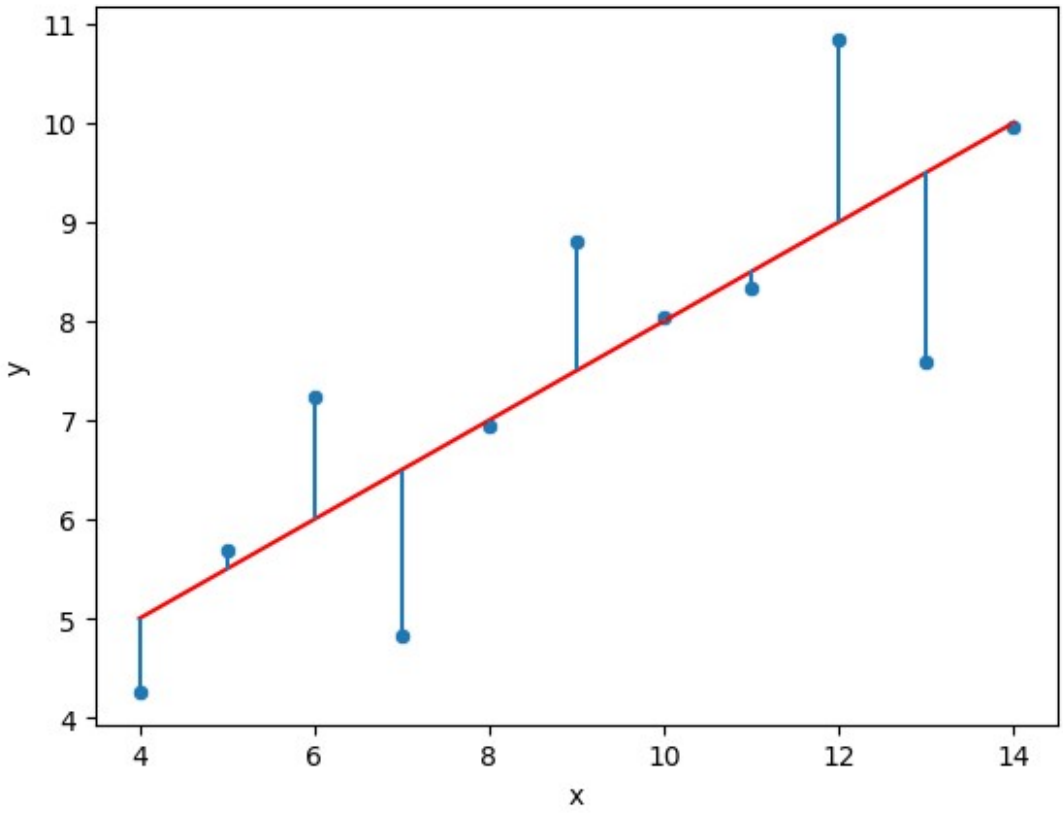
Here are few plots or data where mean_absolute_error , mean_square_error doesnt show right value

Revisiting Anscombe's Quartet: https://en.wikipedia.org/wiki/Anscombe%27s_quartet

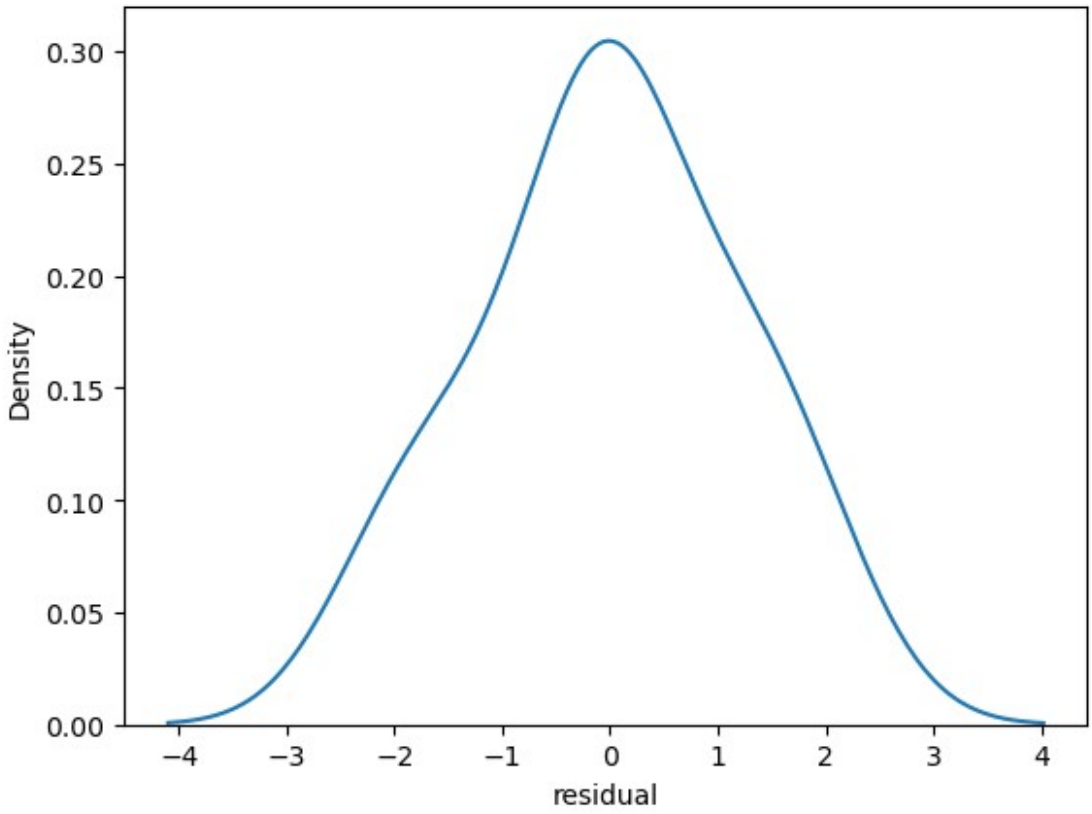
```
quartet = pd.read_csv("D:\\Study\\Programming\\python\\Python course
from udemy\\Udemy - 2022 Python for Machine Learning & Data Science
Masterclass\\01 - Introduction to Course\\1UNZIP-FOR-NOTEBOOKS-FINAL\\
08-Linear-Regression-Models\\anscombes_quartet1.csv")
quartet.head()
```

```
      x     y
0  10.0  8.04
1   8.0  6.95
2  13.0  7.58
3   9.0  8.81
4  11.0  8.33
```

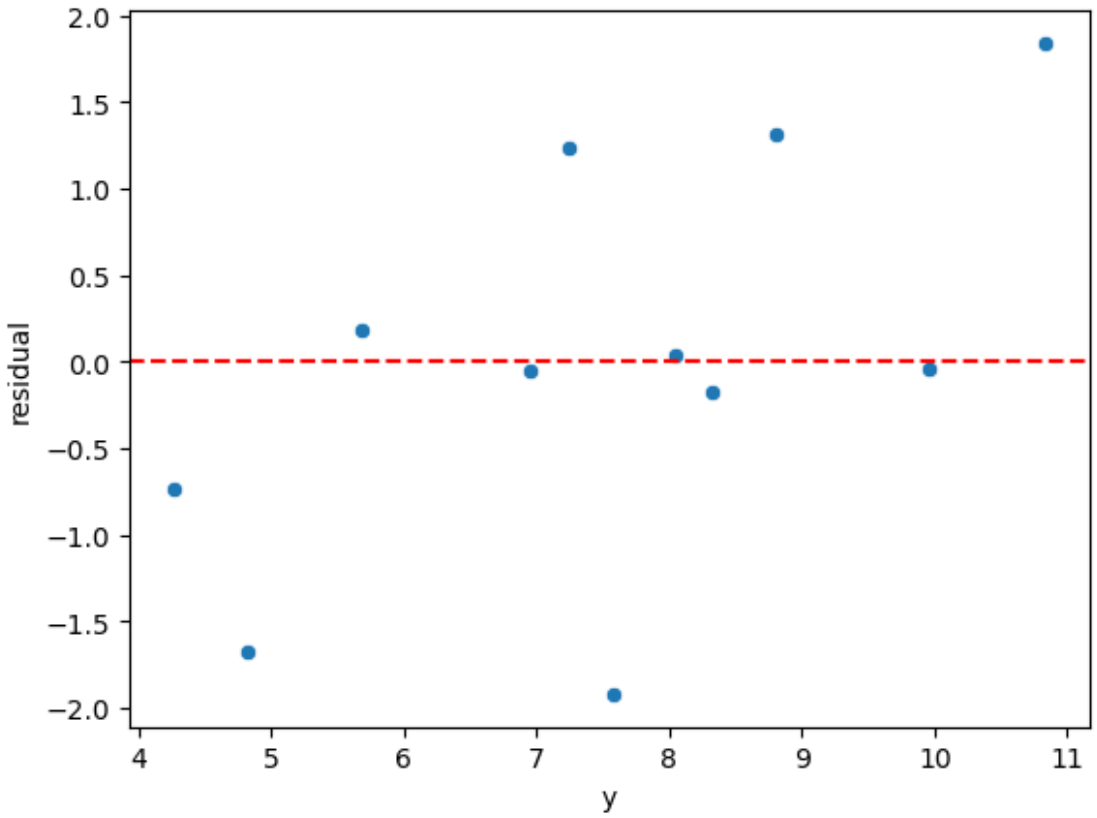
```
# y = 3.00 + 0.500x
quartet['pred_y'] = 3 + 0.5 * quartet['x']
quartet['residual'] = quartet['y'] - quartet['pred_y']

sns.scatterplot(data=quartet,x='x',y='y')
sns.lineplot(data=quartet,x='x',y='pred_y',color='red')
plt.vlines(quartet['x'],quartet['y'],quartet['y']-
quartet['residual']);
```




```
sns.kdeplot(quartet['residual']);
```



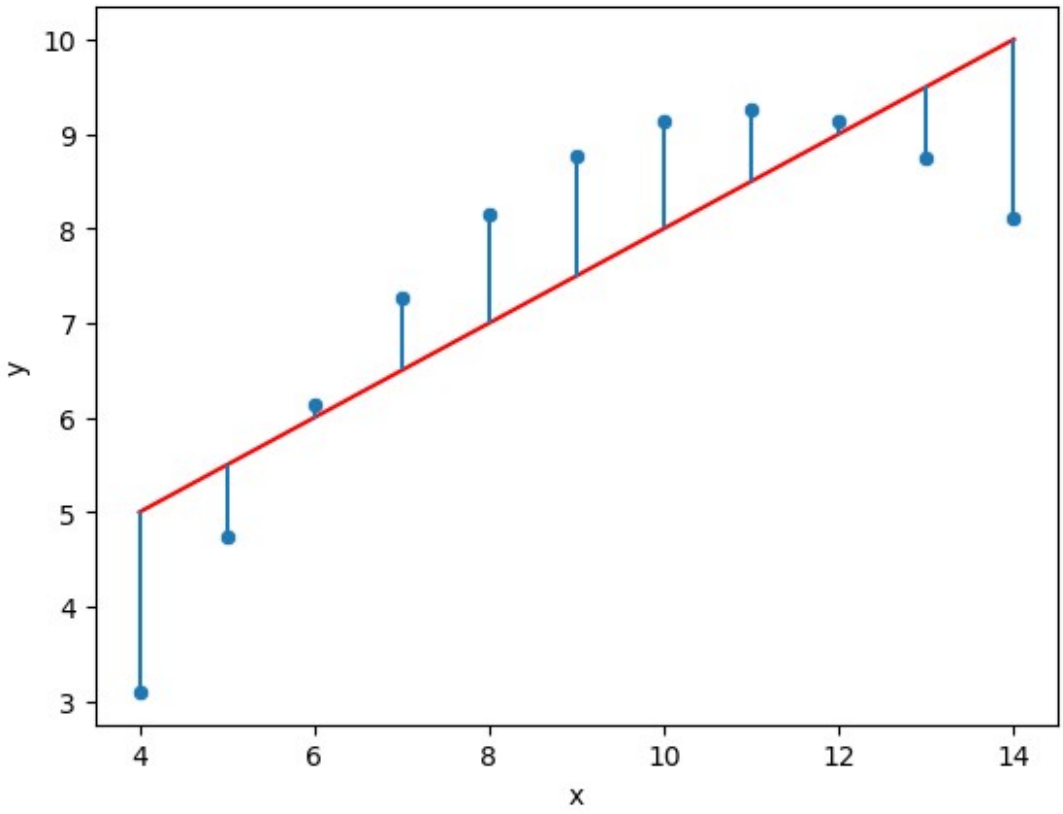

```
sns.scatterplot(data=quartet,x='y',y='residual')
plt.axhline(y=0, color='r', linestyle='--');
```



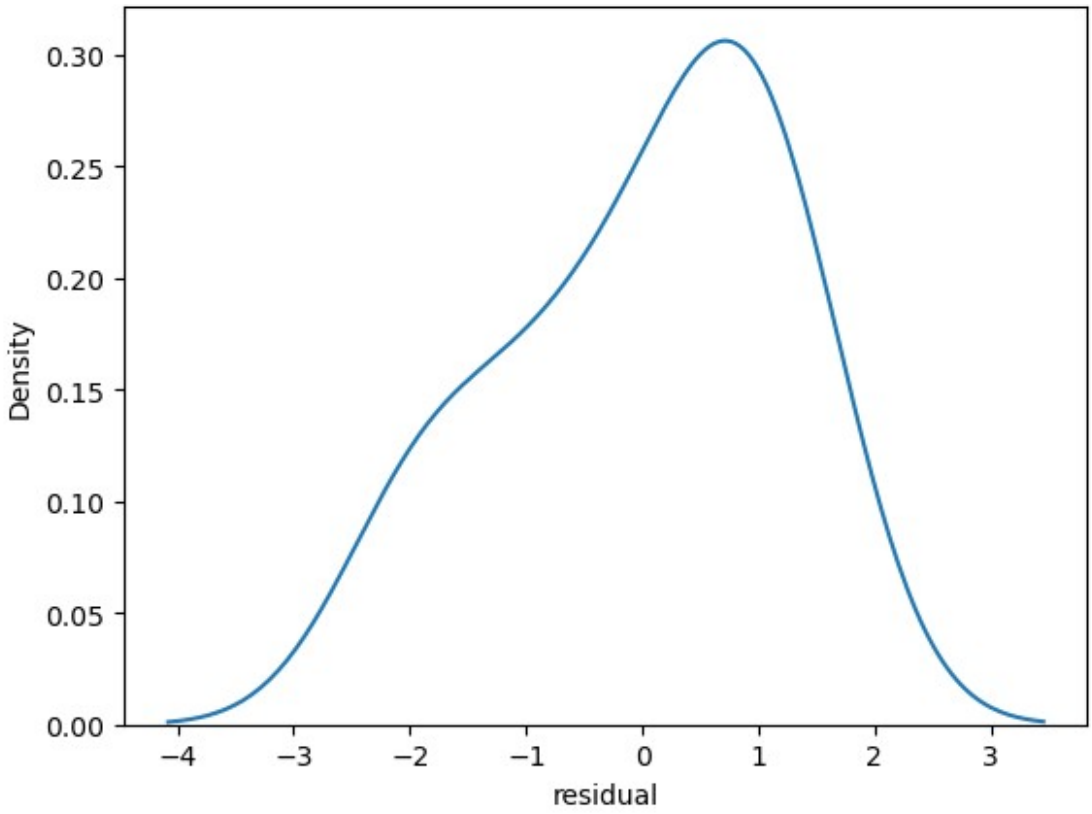

```
quartet = pd.read_csv("D:\\Study\\Programming\\python\\Python course
from udemy\\Udemy - 2022 Python for Machine Learning & Data Science
Masterclass\\01 - Introduction to Course\\1UNZIP-FOR-NOTEBOOKS-FINAL\\
08-Linear-Regression-Models\\anscombes_quartet2.csv")
quartet.head()
```

```
      x     y
0  10.0  9.14
1   8.0  8.14
2  13.0  8.74
3   9.0  8.77
4  11.0  9.26
```

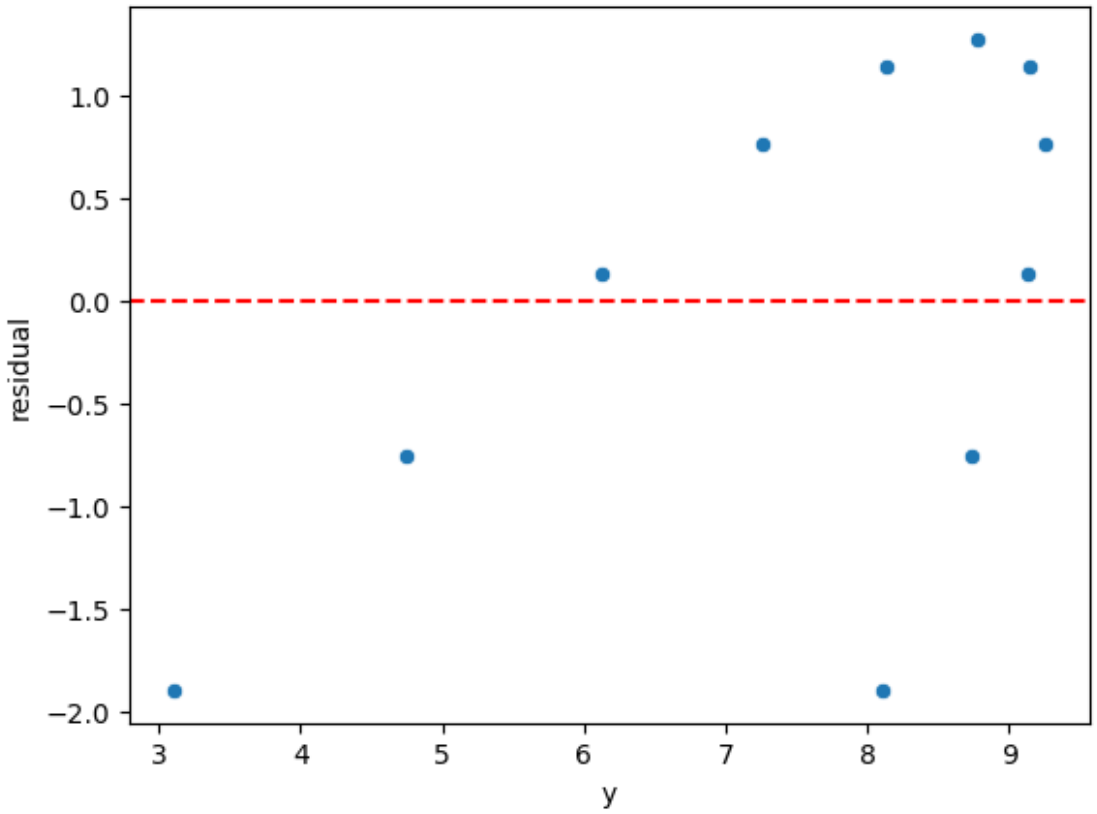
```
quartet.columns = ['x','y']

# y = 3.00 + 0.500x
quartet['pred_y'] = 3 + 0.5 * quartet['x']
quartet['residual'] = quartet['y'] - quartet['pred_y']

sns.scatterplot(data=quartet,x='x',y='y')
sns.lineplot(data=quartet,x='x',y='pred_y',color='red')
plt.vlines(quartet['x'],quartet['y'],quartet['y']-
quartet['residual']);
```




```
sns.kdeplot(quartet['residual']);
```



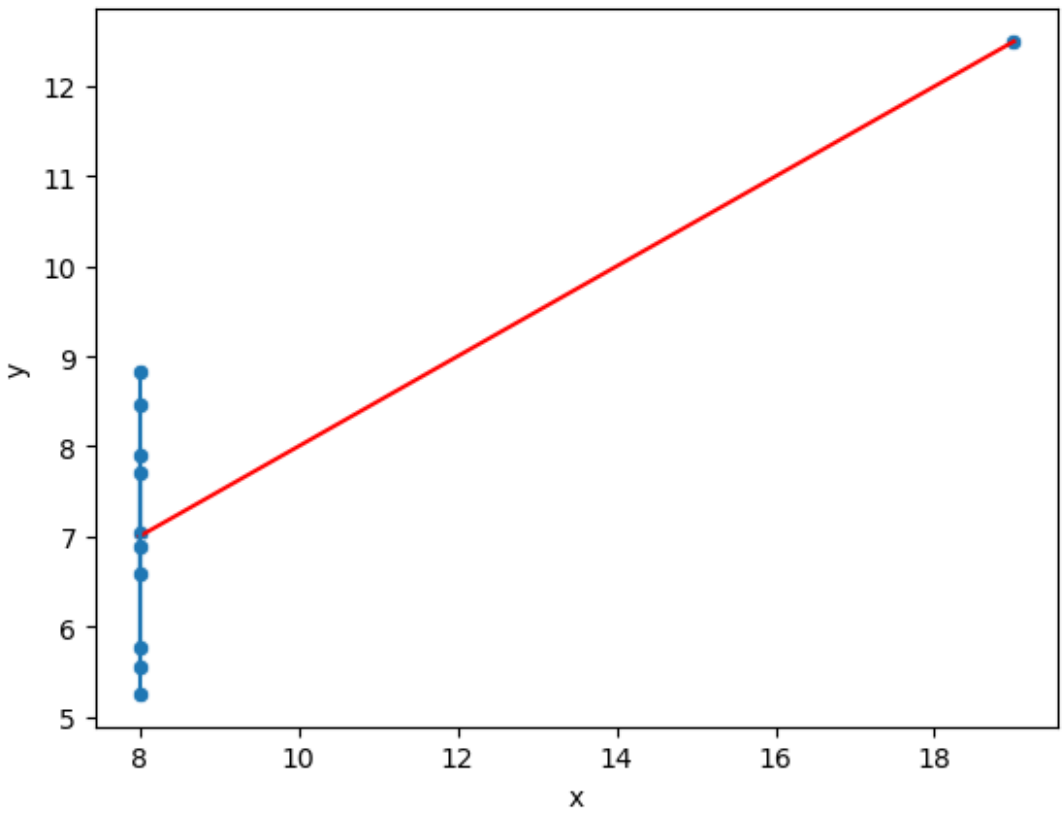

```
sns.scatterplot(data=quartet,x='y',y='residual')
plt.axhline(y=0, color='r', linestyle='--');
```



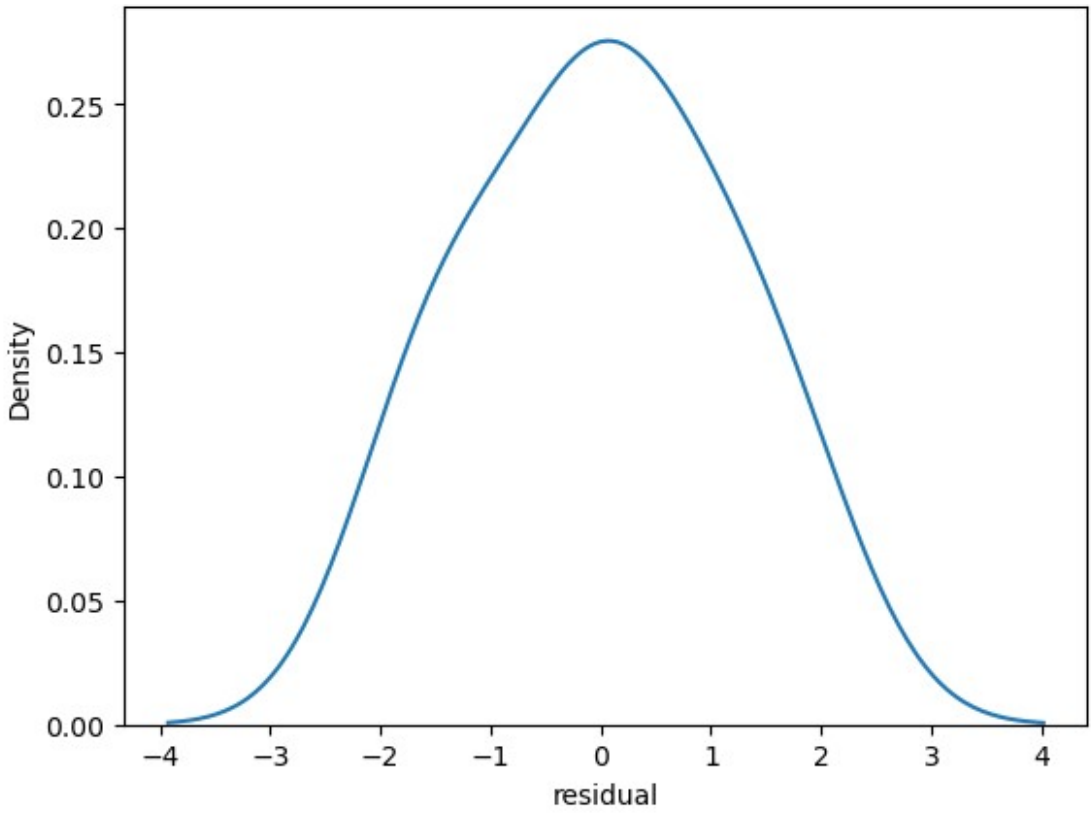

```
quartet = pd.read_csv("D:\\Study\\Programming\\python\\Python course
from udemy\\Udemy - 2022 Python for Machine Learning & Data Science
Masterclass\\01 - Introduction to Course\\1UNZIP-FOR-NOTEBOOKS-FINAL\\
08-Linear-Regression-Models\\anscombes_quartet4.csv")
quartet.head()
```

```
     x     y
0  8.0  6.58
1  8.0  5.76
2  8.0  7.71
3  8.0  8.84
4  8.0  8.47
```

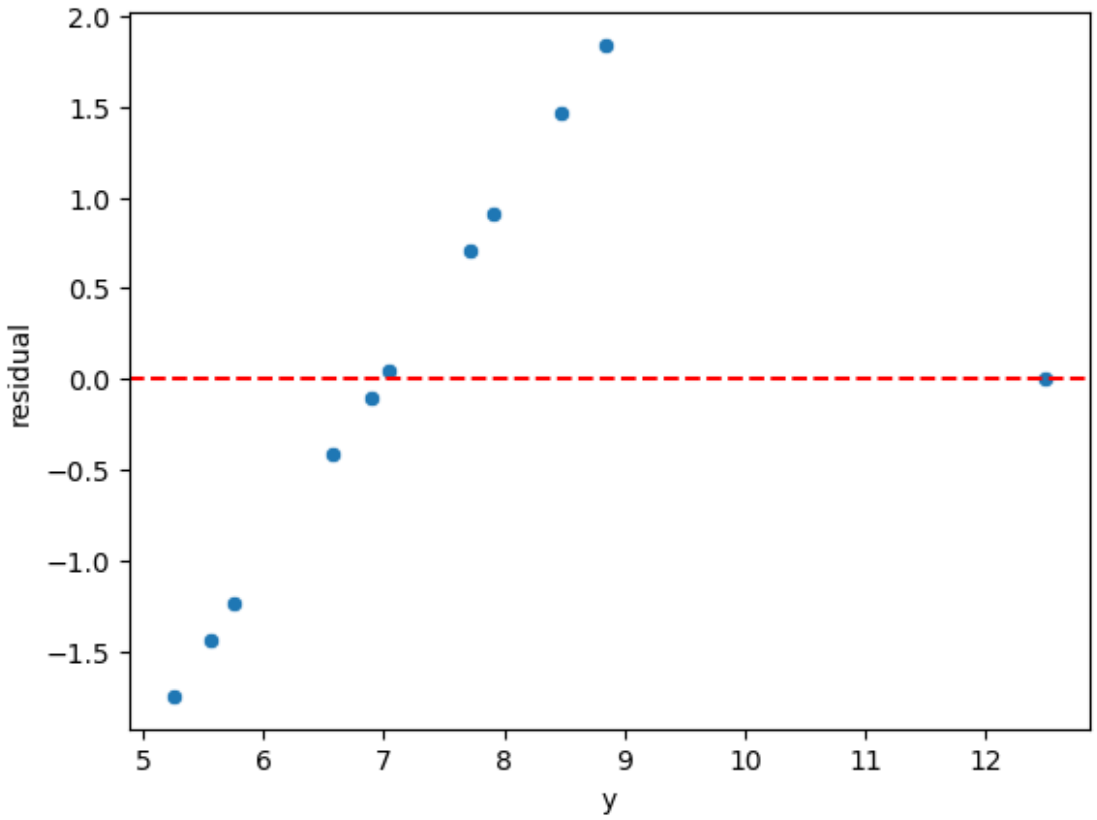
```
# y = 3.00 + 0.500x
quartet['pred_y'] = 3 + 0.5 * quartet['x']

quartet['residual'] = quartet['y'] - quartet['pred_y']

sns.scatterplot(data=quartet,x='x',y='y')
sns.lineplot(data=quartet,x='x',y='pred_y',color='red')
plt.vlines(quartet['x'],quartet['y'],quartet['y']-
quartet['residual']);
```




```
sns.kdeplot(quartet['residual']);
```




```
sns.scatterplot(data=quartet,x='y',y='residual')
plt.axhline(y=0, color='r', linestyle='--');
```



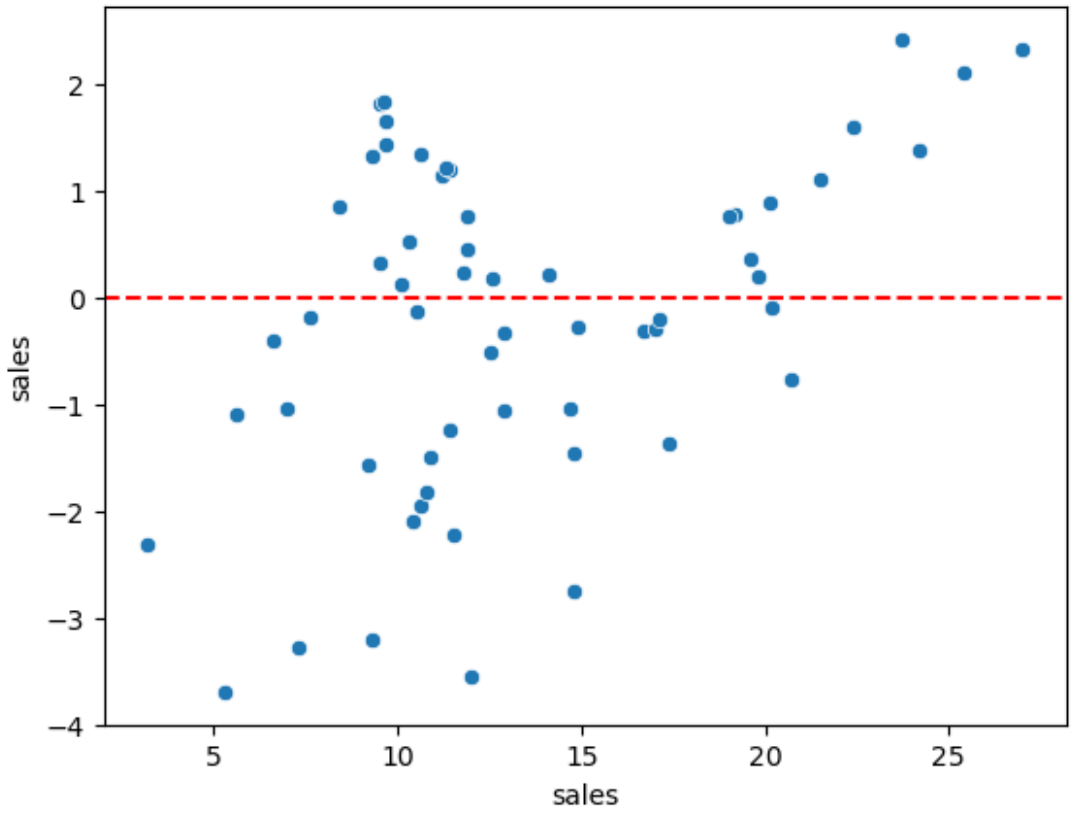
Plotting Residuals

It's also important to plot out residuals and check for normal distribution, this helps us understand if Linear Regression was a valid model choice.

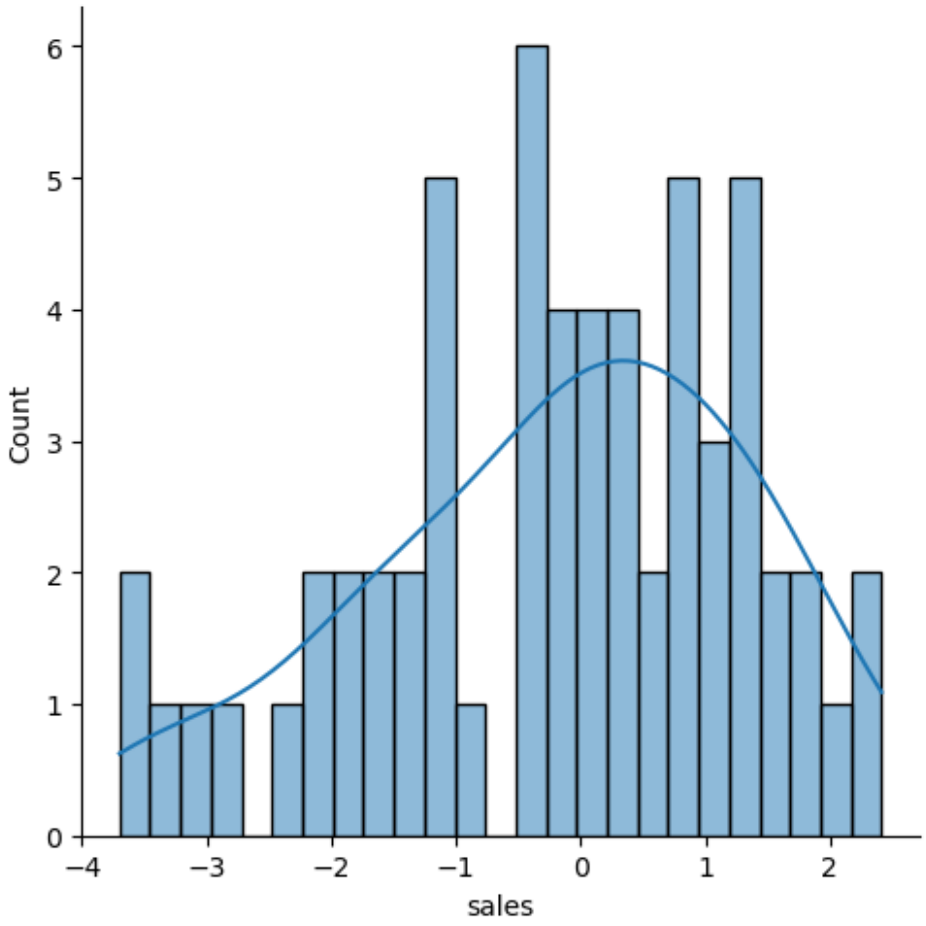
```
# If our model was perfect, these would all be zeros
test_res = Y_test - test_predictions

# Here we are going to see that error
test_residules=Y_test-test_predictions
test_residules[0:10]
```

```
37    -1.041313
109    0.189374
31     0.451111
89    -0.308198
66     0.327143
119   -0.412483
54    -0.089925
74    -0.299540
145    0.524155
142    0.878058
Name: sales, dtype: float64
```

```
# Here we are going to plot that on scatterplot
sns.scatterplot(x=Y_test,y=test_residules)
plt.axhline(y=0, color='red', linestyle='--');
```

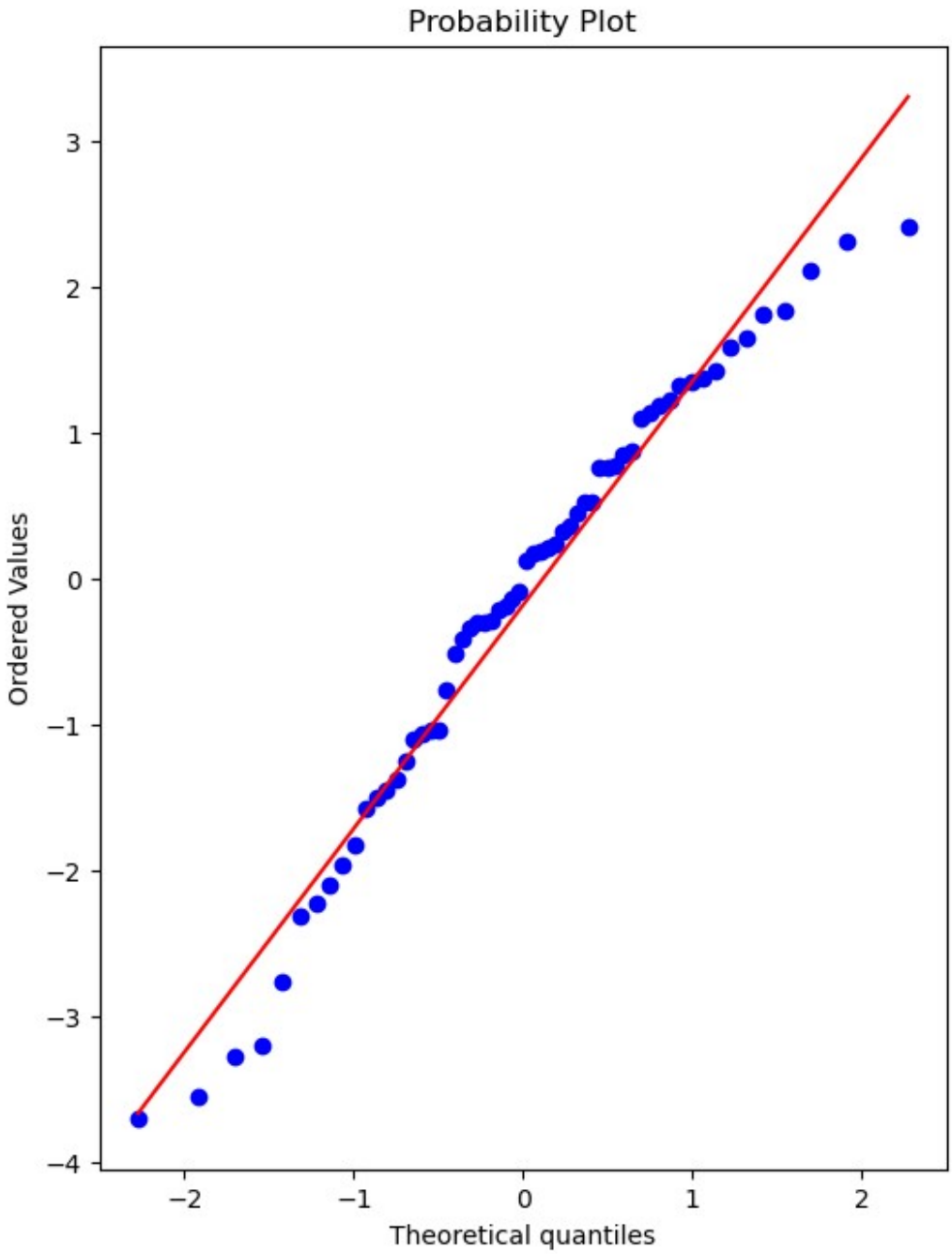



```
# Here to check displot to see error values
sns.displot(test_residules,kde=True,bins=25);
```



Still unsure if normality is a reasonable approximation? We can check against the normal probability plot.

```
# To see what actual look like
import scipy as sp

# Create a figure and axis to plot on
fig, ax = plt.subplots(figsize=(6,8),dpi=100)
# probplot returns the raw values if needed
# we just want to see the plot, so we assign these values to _
_ = sp.stats.probplot(test_res,plot=ax)
```



Retraining Model on Full Data

If we're satisfied with the performance on the test data, before deploying our model to the real world, we should retrain on all our data. (If we were not satisfied, we could update parameters or choose another model, something we'll discuss later on).

```
df=pd.read_csv("D:\\Study\\Programming\\python\\Python course from udemy\\Udemy - 2022 Python for Machine Learning & Data Science
```

```
Masterclass\\01 - Introduction to Course\\1UNZIP-FOR-NOTEBOOKS-FINAL\\
08-Linear-Regression-Models\\Advertising.csv")
df.head()
```

```
      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
```

```
X = df.drop('sales',axis=1) #Here we prepare data that we work on
Y = df['sales'] # we seprate sales in different DataFrame that we have
to predict
```

```
final_model = LinearRegression()
```

```
final_model.fit(X,y)
```

```
LinearRegression()
```

Note how it may not really make sense to recalulate RMSE metrics here, since the model has already seen all the data, its not a fair judgement of performance to calculate RMSE on data its already seen, thus the purpose of the previous examination of test performance.

Deployment, Predictions, and Model Attributes

Final Model Fit

Note, we can only do this since we only have 3 features, for any more it becomes unreasonable.

```
X.head()
```

```
      TV  radio  newspaper
0  230.1   37.8       69.2
1   44.5   39.3       45.1
2   17.2   45.9       69.3
3  151.5   41.3       58.5
4  180.8   10.8       58.4
```

Coefficients

```
final_model.coef_
```

```
array([ 0.04576465,  0.18853002, -0.00103749])
```

Here 0.04576465 for TV , 0.18853002 for radio , -0.00103749 for newspaper

Interpreting the coefficients:

- Holding all other features fixed, a 1 unit (A thousand dollars) increase in TV Spend is associated with an increase in sales of 0.045 "sales units", in this case 1000s of units .

- **This basically means that for every $1000 dollars spend on TV Ads, we could expect 45 more units sold.**

- Holding all other features fixed, a 1 unit (A thousand dollars) increase in Radio Spend is associated with an increase in sales of 0.188 "sales units", in this case 1000s of units .

- **This basically means that for every $1000 dollars spend on Radio Ads, we could expect 188 more units sold.**

- Holding all other features fixed, a 1 unit (A thousand dollars) increase in Newspaper Spend is associated with a **decrease** in sales of 0.001 "sales units", in this case 1000s of units .

- **This basically means that for every $1000 dollars spend on Newspaper Ads, we could actually expect to sell 1 less unit. Being so close to 0, this heavily implies that newspaper spend has no real effect on sales.**

Note! In this case all our units were the same for each feature (1 unit = $1000 of ad spend). But in other datasets, units may not be the same, such as a housing dataset could try to predict a sale price with both a feature for number of bedrooms and a feature of total area like square footage. In this case it would make more sense to *normalize* the data, in order to clearly compare features and results. We will cover normalization later on.

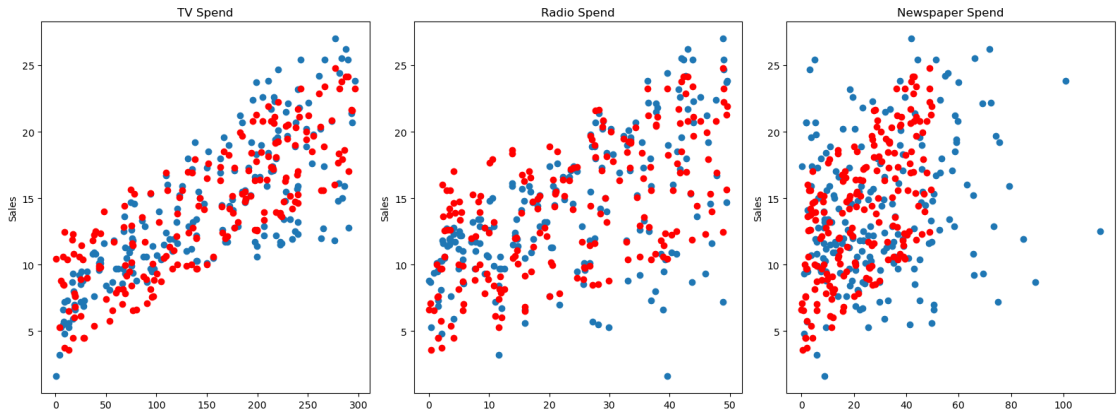
```
df.corr()
```

```
                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
```

```
y_hat = final_model.predict(X)
```

```
fig,axes = plt.subplots(nrows=1,ncols=3,figsize=(16,6))

axes[0].plot(df['TV'],df['sales'],'o')
axes[0].plot(df['TV'],y_hat,'o',color='red')
axes[0].set_ylabel("Sales")
axes[0].set_title("TV Spend")
```

```
axes[1].plot(df['radio'],df['sales'],'o')
axes[1].plot(df['radio'],y_hat,'o',color='red')
axes[1].set_title("Radio Spend")
axes[1].set_ylabel("Sales")

axes[2].plot(df['newspaper'],df['sales'],'o')
axes[2].plot(df['radio'],y_hat,'o',color='red')
axes[2].set_title("Newspaper Spend");
axes[2].set_ylabel("Sales")
plt.tight_layout();
```



Prediction on New Data

Recall , X_test data set looks *exactly* the same as brand new data, so we simply need to call .predict() just as before to predict sales for a new advertising campaign.

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?

```
campaign = [[149,22,12]]
```

```
final_model.predict(campaign)
```

```
C:\Users\Chromsy\AppData\Roaming\Python\Python39\site-packages\
sklearn\base.py:409: UserWarning: X does not have valid feature names,
but LinearRegression was fitted with feature names
  warnings.warn(
```

```
array([13.893032])
```

How accurate is this prediction? No real way to know! We only know truly know our model's performance on the test data, that is why we had to be satisfied by it first, before training our full model

---

Model Persistence (Saving and Loading a Model)

```
# Here is the library that we need for load and save
from joblib import dump, load

# This will save file in system
dump(final_model, "D:\\Study\\sales_model.joblib")
```

```
['D:\\Study\\sales_model.joblib']
```

```
# Load file from system
loaded_model = load("D:\\Study\\sales_model.joblib")

loaded_model.predict(campaign)
```

```
C:\Users\Chromsy\AppData\Roaming\Python\Python39\site-packages\
sklearn\base.py:409: UserWarning: X does not have valid feature names,
but LinearRegression was fitted with feature names
  warnings.warn(

array([13.893032])

loaded_model.coef_

array([ 0.04576465,  0.18853002, -0.00103749])
```

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!

Sample Data

This sample data is from ISLR. It 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.

```
# Import DataFrame file
df= pd.read_csv("D:\\Study\\Programming\\python\\Python course from
udemy\\Udemy - 2022 Python for Machine Learning & Data Science
Masterclass\\01 - Introduction to Course\\1UNZIP-FOR-NOTEBOOKS-FINAL\\
08-Linear-Regression-Models\\Advertising.csv")
df.head()
```

	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

```
# We seprate our DataFrame in X and Y data and then what we have to
predict
X = df.drop('sales', axis=1)
Y = df['sales']
```

SciKit Learn

Polynomial Regression

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].

```
from sklearn.preprocessing import PolynomialFeatures

# Here we choose degree =2 that means we ahve maximum power that will
be 2 and include_bias that will add number too like
# c in y= mx+c and there interaction_only by default is false that
show only extra column like AB , AC , CB not A^2....
polynomial_converter =
PolynomialFeatures(degree=2,include_bias=False )

polynomial_converter.fit(X)

PolynomialFeatures(include_bias=False)

poly_features=polynomial_converter.transform(X)
poly_features
```

```
array([[ 230.1 ,   37.8 ,   69.2 , ..., 1428.84, 2615.76, 4788.64],
       [  44.5 ,   39.3 ,   45.1 , ..., 1544.49, 1772.43, 2034.01],
       [  17.2 ,   45.9 ,   69.3 , ..., 2106.81, 3180.87, 4802.49],
       ...,
       [ 177.  ,    9.3 ,    6.4 , ...,   86.49,   59.52,   40.96],
```

```
       [ 283.6 ,   42.  ,   66.2 , ..., 1764.  , 2780.4 , 4382.44],
       [ 232.1 ,    8.6 ,    8.7 , ...,   73.96,   74.82,   75.69]])
```

```
# Here we can see it transform over Dataframe and now we have degree 2
that means
poly_features.shape
```

```
(200, 9)
```

```
X.iloc[0]
```

```
TV           230.1
radio         37.8
newspaper     69.2
Name: 0, dtype: float64
```

```
poly_features[0]
```

```
array([2.301000e+02, 3.780000e+01, 6.920000e+01, 5.294601e+04,
       8.697780e+03, 1.592292e+04, 1.428840e+03, 2.615760e+03,
       4.788640e+03])
```

here we can see first 3 terms are as it is and then then we have AB,AC,CB then we have A^2,B^2,C^2

```
# If we wanna do fit and transform both together then we can use this
polynomial_converter.fit_transform(X)
```

```
array([[ 230.1 ,   37.8 ,   69.2 , ..., 1428.84, 2615.76, 4788.64],
       [  44.5 ,   39.3 ,   45.1 , ..., 1544.49, 1772.43, 2034.01],
       [  17.2 ,   45.9 ,   69.3 , ..., 2106.81, 3180.87, 4802.49],
       ...,
       [ 177.  ,    9.3 ,    6.4 , ...,   86.49,   59.52,   40.96],
       [ 283.6 ,   42.  ,   66.2 , ..., 1764.  , 2780.4 , 4382.44],
       [ 232.1 ,    8.6 ,    8.7 , ...,   73.96,   74.82,   75.69]])
```

```
poly_features[0][:3]
```

```
array([230.1,  37.8,  69.2])
```

```
poly_features[0][:3]**2
```

```
array([52946.01,  1428.84,  4788.64])
```

The interaction terms

$$x_1 \cdot x_2 \text{ and } x_1 \cdot x_3 \text{ and } x_2 \cdot x_3$$

```
230.1*37.8
```

```
8697.779999999999
```

```
230.1*69.2
```

```
15922.92
```

```
37.8*69.2
```

```
2615.7599999999998
```

Train | Test Split

Make sure you have watched the Machine Learning Overview videos on Supervised Learning to understand why we do this step

```
from sklearn.model_selection import train_test_split
```

```
# random_state:
# https://stackoverflow.com/questions/28064634/random-state-pseudo-random-number-in-scikit-learn
X_train, X_test, Y_train, Y_test = train_test_split(poly_features,Y, test_size=0.3,random_state=101)
# Here test_size is percent of data we want i your test
# random_state is like seed in numpy, to fix random number on every time we hit same seed number
```

Model for fitting on Polynomial Data

Create an instance of the model with parameters

```
from sklearn.linear_model import LinearRegression

model = LinearRegression()
```

Fit/Train the Model on the training data

Make sure you only fit to the training data, in order to fairly evaluate your model's performance on future data

```
model.fit(X_train,Y_train)
```

```
LinearRegression()
```

```
test_prediction = model.predict(X_test)
```

```
model.coef_
```

```
array([ 5.17095811e-02,  1.30848864e-02,  1.20000085e-02, -1.10892474e-04,
        1.14212673e-03, -5.24100082e-05,  3.34919737e-05,  1.46380310e-04,
       -3.04715806e-05])
```

Evaluation on the Test Set

Calculate Performance on Test Set

We want to fairly evaluate our model, so we get performance metrics on the test set (data the model has never seen before).

```
from sklearn.metrics import mean_absolute_error,mean_squared_error
```

```
MAE = mean_absolute_error(Y_test,test_prediction)
MAE
```

```
0.4896798044803811
```

```
MSE= mean_squared_error(Y_test,test_prediction)
MSE
```

```
0.4417505510403745
```

```
RMSE = np.sqrt(MSE)
RMSE
```

```
0.6646431757269268
```

```
df['sales'].mean()
```

```
14.022500000000003
```

Comparison with Simple Linear Regression

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

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

Choosing a Model

Adjusting Parameters

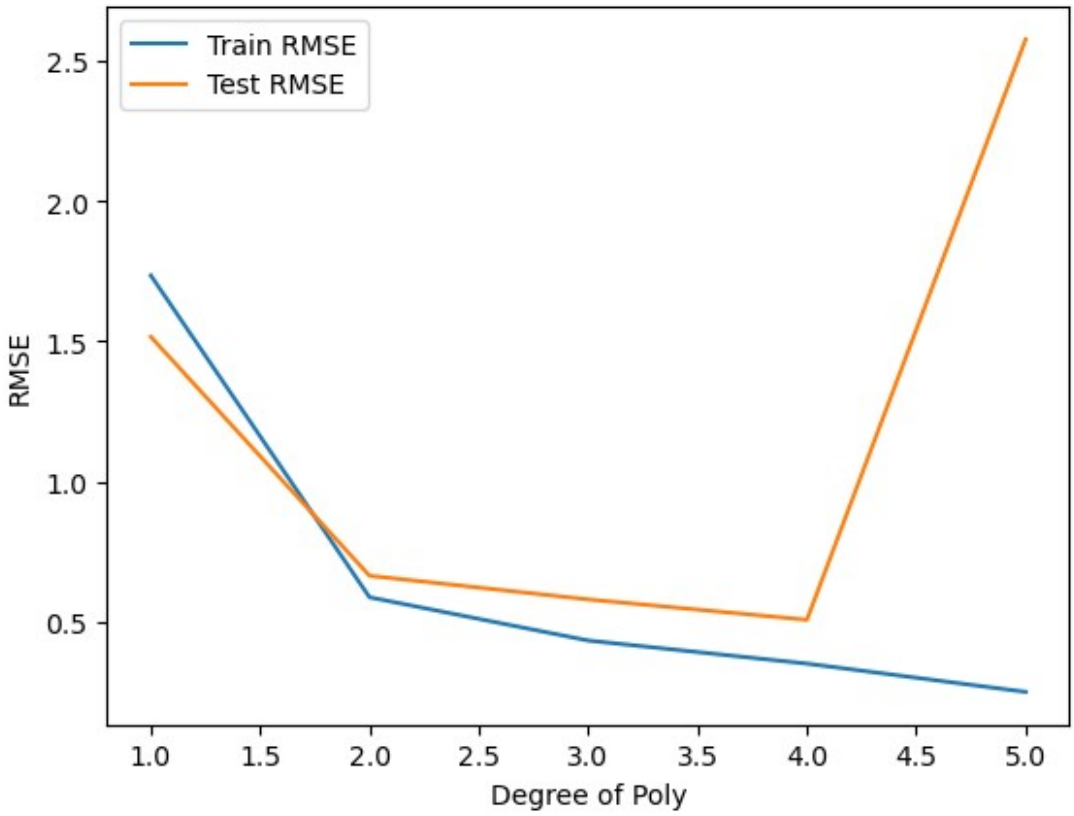
Are we satisfied with this performance? Perhaps a higher order would improve performance even more! But how high is too high? It is now up to us to possibly go back and adjust our model and parameters, let's explore higher order Polynomials in a loop and plot out their error. This will nicely lead us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

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

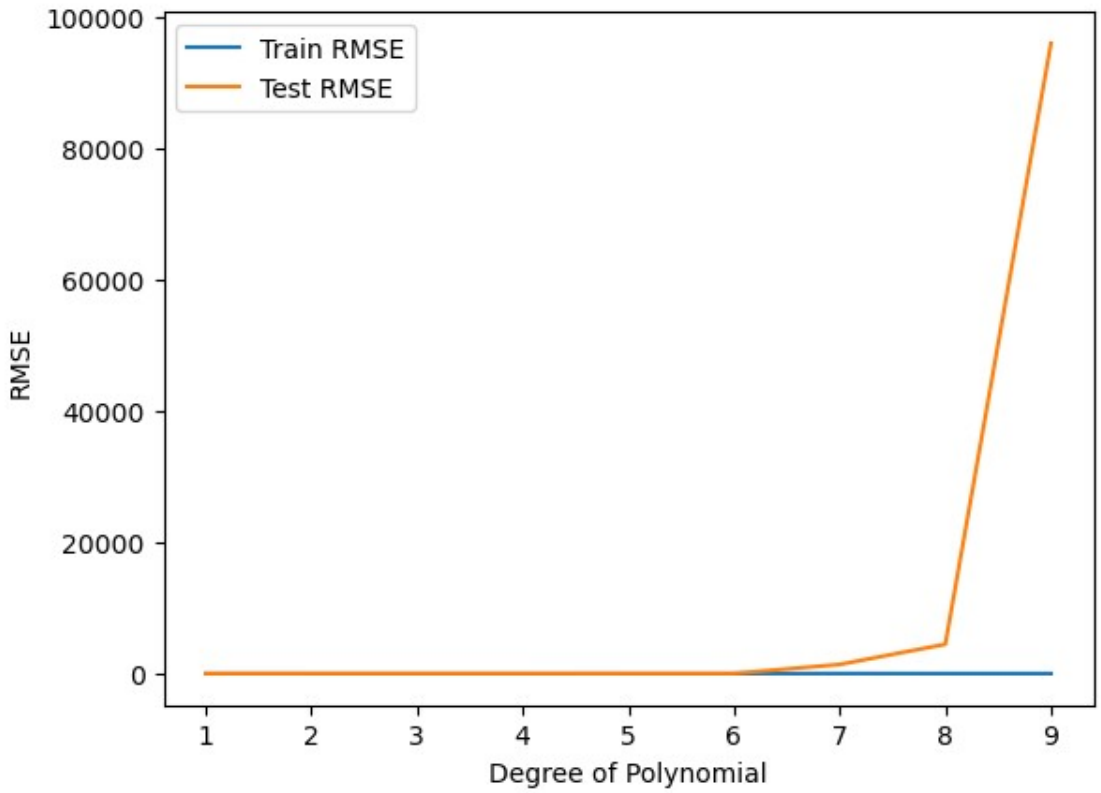
1. Create different order polynomial X data
2. Split that polynomial data for train/test
3. Fit on the training data
4. Report back the metrics on *both* the train and test results
5. Plot these results and explore overfitting

```
from sklearn.preprocessing import PolynomialFeatures

# On text we are going to repeat all above steps but for train and
test data

# TRAINING ERROR PER DEGREE
train_rmse_error = []
# TEST ERROR PER DEGREE
test_rmse_error = []

for d in range(1,10):

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

    # SPLIT THIS NEW POLY DATA SET
    X_train, X_test, Y_train, Y_test =
train_test_split(poly_features,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_error.append(train_rmse)
    test_rmse_error.append(test_rmse)


train_rmse_error
```

```
[1.7345941243293763,
 0.587957408529223,
 0.43393443569020673,
 0.35170836883993467,
 0.25093429467703415,
 0.19712640340673274,
 5.421420423901486,
 0.14180399863580023,
 0.16654350003388185]

test_rmse_error

[1.5161519375993877,
 0.6646431757269268,
 0.5803286825165035,
 0.5077742648623355,
 2.575831205082368,
 4.492668770849738,
 1381.4043738479102,
 4449.599764768951,
 95891.24543764142]

# Here we plot x=1 to 5 which are degree of Polynomial and y on error
values
# Here we take only Degree for 5
plt.plot(range(1,6),train_rmse_error[:5],label='Train RMSE')
plt.plot(range(1,6),test_rmse_error[:5],label='Test RMSE')
plt.xlabel("Degree of Poly")
plt.ylabel("RMSE")
plt.legend();
```




```
# Here we plot x=1 to 10 which are degree of Polynomial and y on error
values
# Herer we take Degree for all that is 10 so we no need to limit in
rmse_error
plt.plot(range(1,10),train_rmse_error, label='Train RMSE')
plt.plot(range(1,10),test_rmse_error, label='Test RMSE')
plt.xlabel("Degree of Polynomial")
plt.ylabel("RMSE")
plt.legend();
```



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

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

final_model = LinearRegression()

# Here we fit both at once here we transform our data(X) and Y because
it final
# We are full satify with degree = 3 so we are going to put that on
whole data
final_converted_X = final_poly_converter.fit_transform(X)
final_model.fit(final_converted_X,Y)

LinearRegression()
```

Saving Model and Converter

```
from joblib import dump , load

# Here we save final model in your system`
dump(final_model, "D:\\Study\\Final_model.joblib")
```

```
['D:\\Study\\Final_model.joblib']
```

```
# Here we save PolynomialFeature where we set degree =3
dump(final_poly_converter,'D:\\Study\\Final_converter.joblib')
```

```
['D:\\Study\\Final_converter.joblib']
```

```
final_poly_converter
```

```
PolynomialFeatures(degree=3, include_bias=False)
```

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?

```
loaded_converter = load("D:\\Study\\Final_converter.joblib")
loaded_model = load("D:\\Study\\Final_model.joblib")
```

```
campaign = [[149,22,12]]
```

We cant put that directly because now values are in degree 3 thats means we have to fit and transfer first

```
tranformed_data = loaded_converter.fit_transform(campaign)
tranformed_data
```

```
array([[1.490000e+02, 2.200000e+01, 1.200000e+01, 2.220100e+04,
        3.278000e+03, 1.788000e+03, 4.840000e+02, 2.640000e+02,
        1.440000e+02, 3.307949e+06, 4.884220e+05, 2.664120e+05,
        7.211600e+04, 3.933600e+04, 2.145600e+04, 1.064800e+04,
        5.808000e+03, 3.168000e+03, 1.728000e+03]])
```

```
loaded_model.predict(tranformed_data)
```

```
array([14.64501014])
```
