My first linear model workbook

August 27, 2022

[1]:
```python
import matplotlib.pyplot as plt

import pandas as pd
import numpy as np
```

0.1

[5]:
```python
data = [(0.5,1), (1,3), (2,2), (3,5), (4,7), (5,8), (6,8), (7,9), (8,10),␣
↪(9,12)]

data = pd.DataFrame(data, columns = ['size', 'price'])

data.head()
```

[5]:

	size	price
0	0.5	1
1	1.0	3
2	2.0	2
3	3.0	5
4	4.0	7

[6]:
```python
data['size']
```

[6]:
```
0    0.5
1    1.0
2    2.0
3    3.0
4    4.0
5    5.0
6    6.0
7    7.0
8    8.0
9    9.0
Name: size, dtype: float64
```

[7]:
```python
# iloc(row, column)      : all
data.iloc[:,0]
```

[7]:
```
0    0.5
1    1.0
2    2.0
3    3.0
4    4.0
5    5.0
6    6.0
7    7.0
8    8.0
9    9.0
Name: size, dtype: float64
```

[8]:
```
data.iloc[:,1]
```

[8]:
```
0     1
1     3
2     2
3     5
4     7
5     8
6     8
7     9
8    10
9    12
Name: price, dtype: int64
```

0.2 scatter plot

[9]:
```
help(plt.scatter)
```

```
Help on function scatter in module matplotlib.pyplot:

scatter(x, y, s=None, c=None, marker=None, cmap=None, norm=None, vmin=None,
vmax=None, alpha=None, linewidths=None, verts=None, edgecolors=None, *,
plotnonfinite=False, data=None, **kwargs)
    A scatter plot of *y* vs *x* with varying marker size and/or color.

    Parameters
    ----------
    x, y : array_like, shape (n, )
        The data positions.

    s : scalar or array_like, shape (n, ), optional
        The marker size in points**2.
        Default is ``rcParams['lines.markersize'] ** 2``.

    c : color, sequence, or sequence of color, optional
        The marker color. Possible values:
```

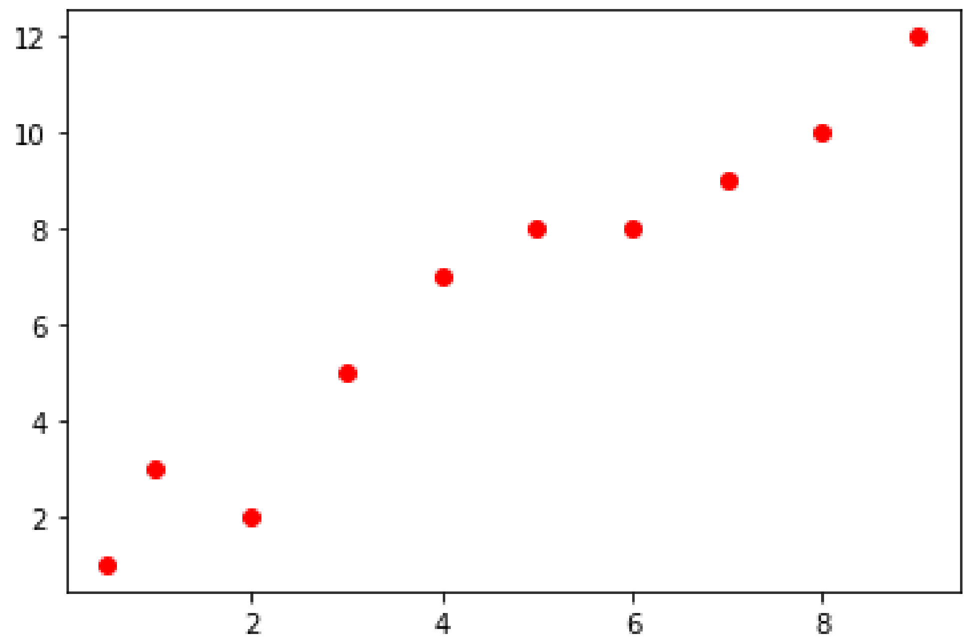
```
    - A single color format string.
    - A sequence of color specifications of length n.
    - A sequence of n numbers to be mapped to colors using *cmap* and
      *norm*.
    - A 2-D array in which the rows are RGB or RGBA.

    Note that *c* should not be a single numeric RGB or RGBA sequence
    because that is indistinguishable from an array of values to be
    colormapped. If you want to specify the same RGB or RGBA value for
    all points, use a 2-D array with a single row.  Otherwise, value-
    matching will have precedence in case of a size matching with *x*
    and *y*.

    Defaults to ``None``. In that case the marker color is determined
    by the value of ``color``, ``facecolor`` or ``facecolors``. In case
    those are not specified or ``None``, the marker color is determined
    by the next color of the ``Axes``' current "shape and fill" color
    cycle. This cycle defaults to :rc:`axes.prop_cycle`.

marker : `~matplotlib.markers.MarkerStyle`, optional
    The marker style. *marker* can be either an instance of the class
    or the text shorthand for a particular marker.
    Defaults to ``None``, in which case it takes the value of
    :rc:`scatter.marker` = 'o'.
    See `~matplotlib.markers` for more information about marker styles.

cmap : `~matplotlib.colors.Colormap`, optional, default: None
    A `.Colormap` instance or registered colormap name. *cmap* is only
    used if *c* is an array of floats. If ``None``, defaults to rc
    ``image.cmap``.

norm : `~matplotlib.colors.Normalize`, optional, default: None
    A `.Normalize` instance is used to scale luminance data to 0, 1.
    *norm* is only used if *c* is an array of floats. If *None*, use
    the default `.colors.Normalize`.

vmin, vmax : scalar, optional, default: None
    *vmin* and *vmax* are used in conjunction with *norm* to normalize
    luminance data. If None, the respective min and max of the color
    array is used. *vmin* and *vmax* are ignored if you pass a *norm*
    instance.

alpha : scalar, optional, default: None
    The alpha blending value, between 0 (transparent) and 1 (opaque).

linewidths : scalar or array_like, optional, default: None
    The linewidth of the marker edges. Note: The default *edgecolors*
```

```
        is 'face'. You may want to change this as well.
        If *None*, defaults to rcParams ``lines.linewidth``.

    edgecolors : {'face', 'none', *None*} or color or sequence of color,
optional.
        The edge color of the marker. Possible values:

        - 'face': The edge color will always be the same as the face color.
        - 'none': No patch boundary will be drawn.
        - A Matplotlib color or sequence of color.

        Defaults to ``None``, in which case it takes the value of
        :rc:`scatter.edgecolors` = 'face'.

        For non-filled markers, the *edgecolors* kwarg is ignored and
        forced to 'face' internally.

    plotnonfinite : boolean, optional, default: False
        Set to plot points with nonfinite *c*, in conjunction with
        `~matplotlib.colors.Colormap.set_bad`.

    Returns
    -------
    paths : `~matplotlib.collections.PathCollection`

    Other Parameters
    ----------------
    **kwargs : `~matplotlib.collections.Collection` properties

    See Also
    --------
    plot : To plot scatter plots when markers are identical in size and
        color.

    Notes
    -----

    * The `.plot` function will be faster for scatterplots where markers
      don't vary in size or color.

    * Any or all of *x*, *y*, *s*, and *c* may be masked arrays, in which
      case all masks will be combined and only unmasked points will be
      plotted.

    * Fundamentally, scatter works with 1-D arrays; *x*, *y*, *s*, and *c*
      may be input as 2-D arrays, but within scatter they will be
      flattened. The exception is *c*, which will be flattened only if its
      size matches the size of *x* and *y*.
```

```
    .. note::
        In addition to the above described arguments, this function can take a
        **data** keyword argument. If such a **data** argument is given, the
        following arguments are replaced by **data[<arg>]**:

        * All arguments with the following names: 'c', 'color', 'edgecolors',
'facecolor', 'facecolors', 'linewidths', 's', 'x', 'y'.

        Objects passed as **data** must support item access (``data[<arg>]``)
and
        membership test (``<arg> in data``).
```

[11]:
```python
x = data['size']
y = data['price']

plt.scatter(x,y,color = "red")
```

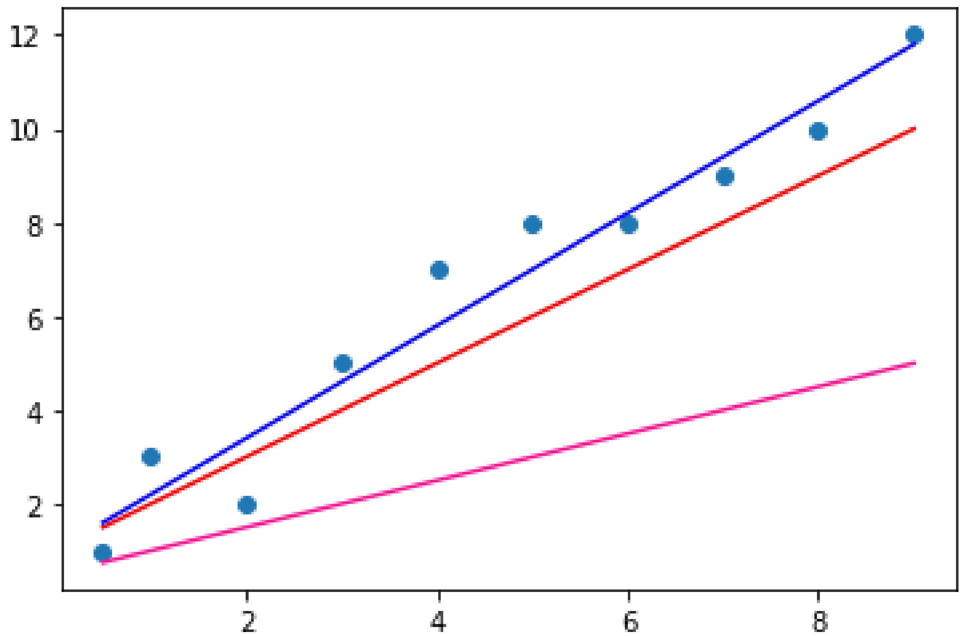
[11]:
```
<matplotlib.collections.PathCollection at 0x28a4319ec08>
```



[14]:
```python
beta0 = 1
beta1 = 1

fitted_y = 1 + 1*x
```

```
plt.scatter(x,y)
plt.plot(x, fitted_y, color = 'red')

fitted_y2 = 1 + 1.2*x
plt.plot(x, fitted_y2, color = 'blue')


fitted_y3 = 0.5 + 0.5*x
plt.plot(x, fitted_y3, color = "deeppink"'')
```

[14]: [<matplotlib.lines.Line2D at 0x28a4948f348>]



[8]:
```
fitted_y
```

[8]:
```
0     1.5
1     2.0
2     3.0
3     4.0
4     5.0
5     6.0
6     7.0
7     8.0
8     9.0
9    10.0
Name: size, dtype: float64
```

[15]:
```python
residuals = y - fitted_y
residuals
```

[15]:
```
0   -0.5
1    1.0
2   -1.0
3    1.0
4    2.0
5    2.0
6    1.0
7    1.0
8    1.0
9    2.0
dtype: float64
```

[ ]:
```python
!pip install seaborn
```

[17]:
```python
import seaborn as sns
```

[18]:
```python
help(sns.regplot)
```

```
Help on function regplot in module seaborn.regression:

regplot(x, y, data=None, x_estimator=None, x_bins=None, x_ci='ci', scatter=True,
fit_reg=True, ci=95, n_boot=1000, units=None, seed=None, order=1,
logistic=False, lowess=False, robust=False, logx=False, x_partial=None,
y_partial=None, truncate=True, dropna=True, x_jitter=None, y_jitter=None,
label=None, color=None, marker='o', scatter_kws=None, line_kws=None, ax=None)
    Plot data and a linear regression model fit.

    There are a number of mutually exclusive options for estimating the
    regression model. See the :ref:`tutorial <regression_tutorial>` for more
    information.

    Parameters
    ----------
    x, y: string, series, or vector array
        Input variables. If strings, these should correspond with column names
        in ``data``. When pandas objects are used, axes will be labeled with
        the series name.
    data : DataFrame
        Tidy ("long-form") dataframe where each column is a variable and each
        row is an observation.
    x_estimator : callable that maps vector -> scalar, optional
        Apply this function to each unique value of ``x`` and plot the
        resulting estimate. This is useful when ``x`` is a discrete variable.
        If ``x_ci`` is given, this estimate will be bootstrapped and a
        confidence interval will be drawn.
```

```
x_bins : int or vector, optional
    Bin the ``x`` variable into discrete bins and then estimate the central
    tendency and a confidence interval. This binning only influences how
    the scatterplot is drawn; the regression is still fit to the original
    data.  This parameter is interpreted either as the number of
    evenly-sized (not necessary spaced) bins or the positions of the bin
    centers. When this parameter is used, it implies that the default of
    ``x_estimator`` is ``numpy.mean``.
x_ci : "ci", "sd", int in [0, 100] or None, optional
    Size of the confidence interval used when plotting a central tendency
    for discrete values of ``x``. If ``"ci"``, defer to the value of the
    ``ci`` parameter. If ``"sd"``, skip bootstrapping and show the
    standard deviation of the observations in each bin.
scatter : bool, optional
    If ``True``, draw a scatterplot with the underlying observations (or
    the ``x_estimator`` values).
fit_reg : bool, optional
    If ``True``, estimate and plot a regression model relating the ``x``
    and ``y`` variables.
ci : int in [0, 100] or None, optional
    Size of the confidence interval for the regression estimate. This will
    be drawn using translucent bands around the regression line. The
    confidence interval is estimated using a bootstrap; for large
    datasets, it may be advisable to avoid that computation by setting
    this parameter to None.
n_boot : int, optional
    Number of bootstrap resamples used to estimate the ``ci``. The default
    value attempts to balance time and stability; you may want to increase
    this value for "final" versions of plots.
units : variable name in ``data``, optional
    If the ``x`` and ``y`` observations are nested within sampling units,
    those can be specified here. This will be taken into account when
    computing the confidence intervals by performing a multilevel bootstrap
    that resamples both units and observations (within unit). This does not
    otherwise influence how the regression is estimated or drawn.
seed : int, numpy.random.Generator, or numpy.random.RandomState, optional
    Seed or random number generator for reproducible bootstrapping.
order : int, optional
    If ``order`` is greater than 1, use ``numpy.polyfit`` to estimate a
    polynomial regression.
logistic : bool, optional
    If ``True``, assume that ``y`` is a binary variable and use
    ``statsmodels`` to estimate a logistic regression model. Note that this
    is substantially more computationally intensive than linear regression,
    so you may wish to decrease the number of bootstrap resamples
    (``n_boot``) or set ``ci`` to None.
lowess : bool, optional
    If ``True``, use ``statsmodels`` to estimate a nonparametric lowess
```

```
    model (locally weighted linear regression). Note that confidence
    intervals cannot currently be drawn for this kind of model.
robust : bool, optional
    If ``True``, use ``statsmodels`` to estimate a robust regression. This
    will de-weight outliers. Note that this is substantially more
    computationally intensive than standard linear regression, so you may
    wish to decrease the number of bootstrap resamples (``n_boot``) or set
    ``ci`` to None.
logx : bool, optional
    If ``True``, estimate a linear regression of the form y ~ log(x), but
    plot the scatterplot and regression model in the input space. Note that
    ``x`` must be positive for this to work.
{x,y}_partial : strings in ``data`` or matrices
    Confounding variables to regress out of the ``x`` or ``y`` variables
    before plotting.
truncate : bool, optional
    By default, the regression line is drawn to fill the x axis limits
    after the scatterplot is drawn. If ``truncate`` is ``True``, it will
    instead by bounded by the data limits.
{x,y}_jitter : floats, optional
    Add uniform random noise of this size to either the ``x`` or ``y``
    variables. The noise is added to a copy of the data after fitting the
    regression, and only influences the look of the scatterplot. This can
    be helpful when plotting variables that take discrete values.
label : string
    Label to apply to either the scatterplot or regression line (if
    ``scatter`` is ``False``) for use in a legend.
color : matplotlib color
    Color to apply to all plot elements; will be superseded by colors
    passed in ``scatter_kws`` or ``line_kws``.
marker : matplotlib marker code
    Marker to use for the scatterplot glyphs.
{scatter,line}_kws : dictionaries
    Additional keyword arguments to pass to ``plt.scatter`` and
    ``plt.plot``.
ax : matplotlib Axes, optional
    Axes object to draw the plot onto, otherwise uses the current Axes.

Returns
-------
ax : matplotlib Axes
    The Axes object containing the plot.

See Also
--------
lmplot : Combine :func:`regplot` and :class:`FacetGrid` to plot multiple
         linear relationships in a dataset.
jointplot : Combine :func:`regplot` and :class:`JointGrid` (when used with
```

```
            ``kind="reg"``).
pairplot : Combine :func:`regplot` and :class:`PairGrid` (when used with
           ``kind="reg"``).
residplot : Plot the residuals of a linear regression model.

Notes
-----

The :func:`regplot` and :func:`lmplot` functions are closely related, but
the former is an axes-level function while the latter is a figure-level
function that combines :func:`regplot` and :class:`FacetGrid`.


It's also easy to combine combine :func:`regplot` and :class:`JointGrid` or
:class:`PairGrid` through the :func:`jointplot` and :func:`pairplot`
functions, although these do not directly accept all of :func:`regplot`'s
parameters.

Examples
--------

Plot the relationship between two variables in a DataFrame:

.. plot::
    :context: close-figs

    >>> import seaborn as sns; sns.set(color_codes=True)
    >>> tips = sns.load_dataset("tips")
    >>> ax = sns.regplot(x="total_bill", y="tip", data=tips)

Plot with two variables defined as numpy arrays; use a different color:

.. plot::
    :context: close-figs

    >>> import numpy as np; np.random.seed(8)
    >>> mean, cov = [4, 6], [(1.5, .7), (.7, 1)]
    >>> x, y = np.random.multivariate_normal(mean, cov, 80).T
    >>> ax = sns.regplot(x=x, y=y, color="g")

Plot with two variables defined as pandas Series; use a different marker:

.. plot::
    :context: close-figs

    >>> import pandas as pd
    >>> x, y = pd.Series(x, name="x_var"), pd.Series(y, name="y_var")
    >>> ax = sns.regplot(x=x, y=y, marker="+")
```

Use a 68% confidence interval, which corresponds with the standard error of the estimate, and extend the regression line to the axis limits:

```
.. plot::
    :context: close-figs

    >>> ax = sns.regplot(x=x, y=y, ci=68, truncate=False)
```

Plot with a discrete ``x`` variable and add some jitter:

```
.. plot::
    :context: close-figs

    >>> ax = sns.regplot(x="size", y="total_bill", data=tips, x_jitter=.1)
```

Plot with a discrete ``x`` variable showing means and confidence intervals for unique values:

```
.. plot::
    :context: close-figs

    >>> ax = sns.regplot(x="size", y="total_bill", data=tips,
    …                    x_estimator=np.mean)
```

Plot with a continuous variable divided into discrete bins:

```
.. plot::
    :context: close-figs

    >>> ax = sns.regplot(x=x, y=y, x_bins=4)
```

Fit a higher-order polynomial regression:

```
.. plot::
    :context: close-figs

    >>> ans = sns.load_dataset("anscombe")
    >>> ax = sns.regplot(x="x", y="y", data=ans.loc[ans.dataset == "II"],
    …                    scatter_kws={"s": 80},
    …                    order=2, ci=None)
```

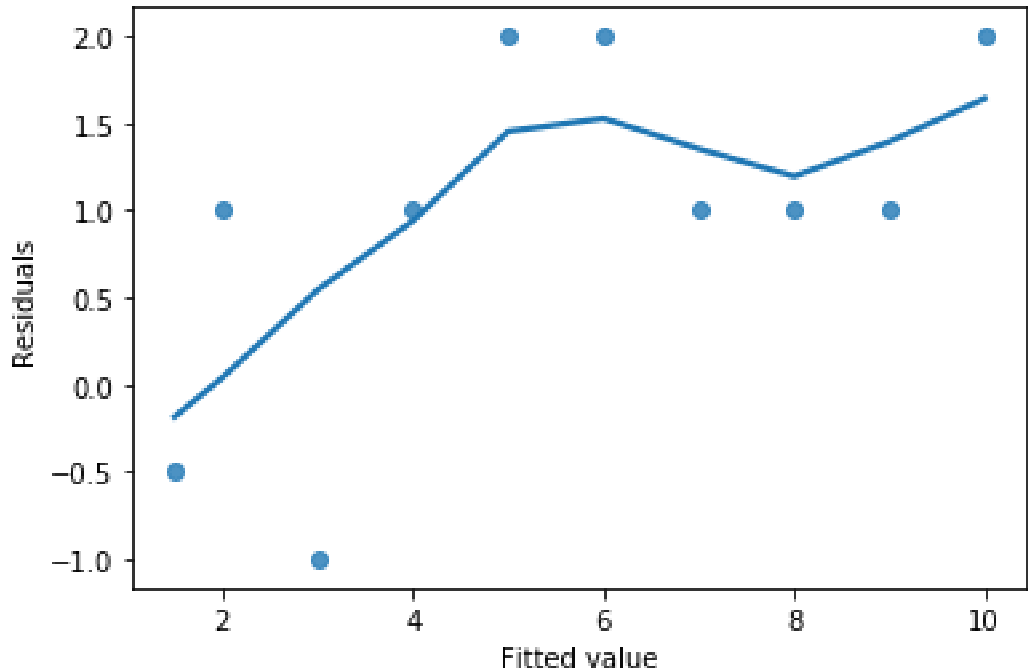
Fit a robust regression and don't plot a confidence interval:

```
.. plot::
    :context: close-figs

    >>> ax = sns.regplot(x="x", y="y", data=ans.loc[ans.dataset == "III"],
```

```
    …                   scatter_kws={"s": 80},
    …                   robust=True, ci=None)

Fit a logistic regression; jitter the y variable and use fewer bootstrap
iterations:

.. plot::
    :context: close-figs

    >>> tips["big_tip"] = (tips.tip / tips.total_bill) > .175
    >>> ax = sns.regplot(x="total_bill", y="big_tip", data=tips,
    …                   logistic=True, n_boot=500, y_jitter=.03)

Fit the regression model using log(x):

.. plot::
    :context: close-figs

    >>> ax = sns.regplot(x="size", y="total_bill", data=tips,
    …                   x_estimator=np.mean, logx=True)
```

[19]:
```python
sns.regplot(fitted_y, residuals,lowess=True)

plt.xlabel("Fitted value")
plt.ylabel("Residuals")
```

[19]:
```
Text(0, 0.5, 'Residuals')
```



[21]:
```
pd.DataFrame({'Resi': residuals,
                            'Fitted_Y': fitted_y})
```

[21]:

	Resi	Fitted_Y
0	-0.5	1.5
1	1.0	2.0
2	-1.0	3.0
3	1.0	4.0
4	2.0	5.0
5	2.0	6.0
6	1.0	7.0
7	1.0	8.0
8	1.0	9.0
9	2.0	10.0