03_LinearRegression

April 20, 2026

1 Linear Regression Notes

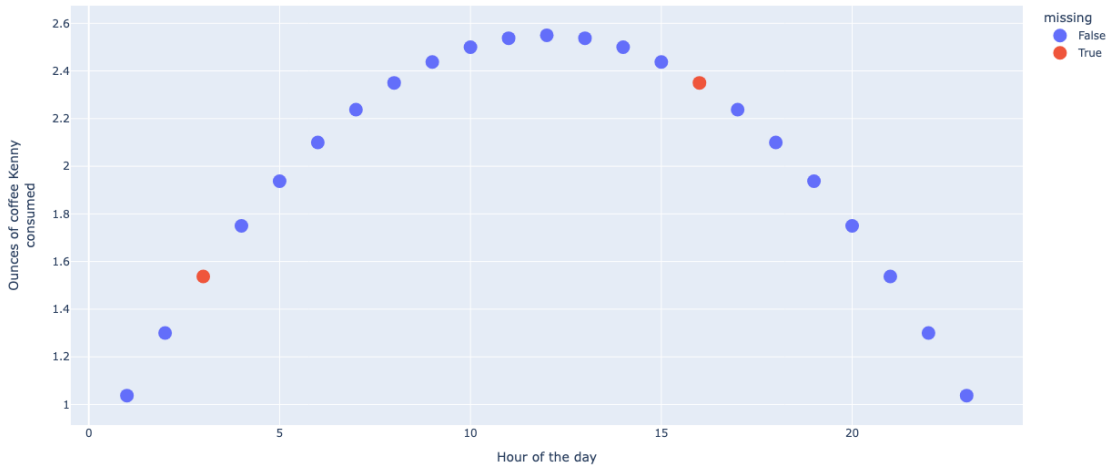
These notes rely on a variety of sources: - Chapters 1-2 of Shalizi's book - Atri's slides from 440 - Varun's prior notes from this class

[6]:
```
import numpy as np
import pandas as pd
import plotly.express as px
import plotly.graph_objects as go
from scipy.stats import norm
import matplotlib.pyplot as plt
import warnings
warnings.simplefilter(action = "ignore", category = FutureWarning)
```

2 A Motivating Example

[3]:
```
np.random.seed(seed=25)
data = pd.DataFrame([(x, (-.5*(x**2)+12*x+30)/40)
                     for x in range(1,24)],
                    columns= ['x','y'])
data['y'] = data['y'] #+ norm(0,.1).rvs(len(data))
missing_data = [False] * len(data)

missing_data[2] = True
missing_data[15] = True
data = data.assign(missing = missing_data)
fig = px.scatter(data, x="x",y='y',
           labels = {"x" : "Hour of the day",
                     "y" : "Ounces of coffee Kenny<br>consumed"},
           color="missing",
          height=600, width=800)
fig.update_traces(marker={'size': 15})
fig.show()
```



2.1 Necessary thought exercise #1

- What is your best guess for the ounces of coffee I consumed at $x = 3$ (3am)?
- What about for $x = 16$ (4pm)?
- **How would you figure that out?**

2.1.1 Why is this exercise necessary?

In Machine Learning, we ask the question of "how would you figure this out" via a **three step process**:

1. We define a **model class** (or synonymously, a **hypotheses class**). We will pick one of those to be our predictive model.
2. We define a **loss function**. This defines the "best" model, i.e. the one we're going to pick.
3. We define an **optimization algorithm**. This is *how we actually find the best model.*

2.1.2 Digging deeper... Model Class

A **model class**, or **hypothesis class**, sometimes represented as $\mathcal{H}$, is the set of all possible functions that we could possibly learn, and select for our final model.

Some desireable properties of a model class: - **Parsimonious representation**: We would like our model class to have its representation size not depend on the size of the dataset. In addition, models with small representation size can be considered to be the "right" model... we'll talk about this later on in class. - **Efficient training**: We have to actually be able to find the best model. - **Efficient prediction**: We don't want to have to wait forever for our model to make predictions! - **Model expressivity**: In other words, for "real life" data, how accurately can this model class predict the correct labels?

2.1.3 Digging deeper... Loss function

From Weinberger's ML course:

> We try to find a function h within the hypothesis class that makes the fewest mistakes within our training data... How can we find the best function? For this we need some way to evaluate what it means for one function to be better than another. This is where the loss function (aka risk function) comes in. A loss function evaluates a hypothesis $h \in \mathcal{H}$ on our training data and tells us how bad it is. The higher the loss, the worse it is - a loss of zero means it makes perfect predictions.

Let's call our loss function (for a given hypothesis) $\mathcal{L}(h)$.

2.1.4 Digging deeper ... Optimization Algorithm

Our goal now is just to find $h^* \in \mathcal{H}$ such that:

$$h^* = \underset{h \in \mathcal{H}}{\operatorname{argmin}} \mathcal{L}(h)$$

3 Motivating Linear Regression

One way to motivate/understand (ordinary least squares) linear regression is that it defines a specific model class, with a specific loss function, and has a closed-form approach to optimization. However, I'm going to take a slightly different motivating approach first, and then circle back to this.

3.1 Necessary thought exercise #2

OK, now... **what if you could only make 1 guess for both of them?**

3.1.1 The "Predict Only One" Best Answer

This section largely follows Shalizi, Chapter 1

As we saw above, to say one model is "better" than another, we need some notion of "better". For reasons that will become clear as we move through this course, a typical notion of "better" is the **mean squared error (MSE)**. For a true value y and a predicted value $\hat{y}$, the squared error of the prediction is: $(y - \hat{y})^2$. The MSE is the mean of this over all y values that we consider (e.g. all y values in our test set).

Exercises: - Why do we square this? - What other things could you think to do instead?

Now, let's assume we're trying to predict the value of some random variable Y, and our goal is to make the best single prediction for it. Let's use MSE as our error metric. Because Y is a random variable, we cannot specify an exact value for this MSE. Instead, we have to think about the *expectation* of that MSE. Let's keep $\hat{y}$ as our prediction. Then our goal is to find the best possible value of $\hat{y}$; let's call (that best possible) value h^* to remind us of the connection to the text above. Put formally, we want to solve the following optimization problem:

$$h^* = \underset{\hat{y} \in \mathbb{R}}{\operatorname{argmin}} MSE(\hat{y}) = \underset{\hat{y} \in \mathbb{R}}{\operatorname{argmin}} \mathbb{E}\left[(Y - \hat{y})^2\right]$$

With some fancy manipulations (see Shalizi), we get that

$$MSE(\hat{y}) = (\mathbb{E}[Y] - \hat{y})^2 + \text{Var}[Y]$$

Now, we want to find the optimal value for $\hat{y}$. Notice that the second term doesn't depend on our prediction, so we don't need to worry about it during optimization. So now, we want to find:

$$h^* = \underset{\hat{y} \in \mathbb{R}}{\text{argmin}} \, (\mathbb{E}[Y] - \hat{y})^2$$

The solution to this is $h^* = \mathbb{E}[Y]$; that is, the best prediction to make is the expected value of the random variable!

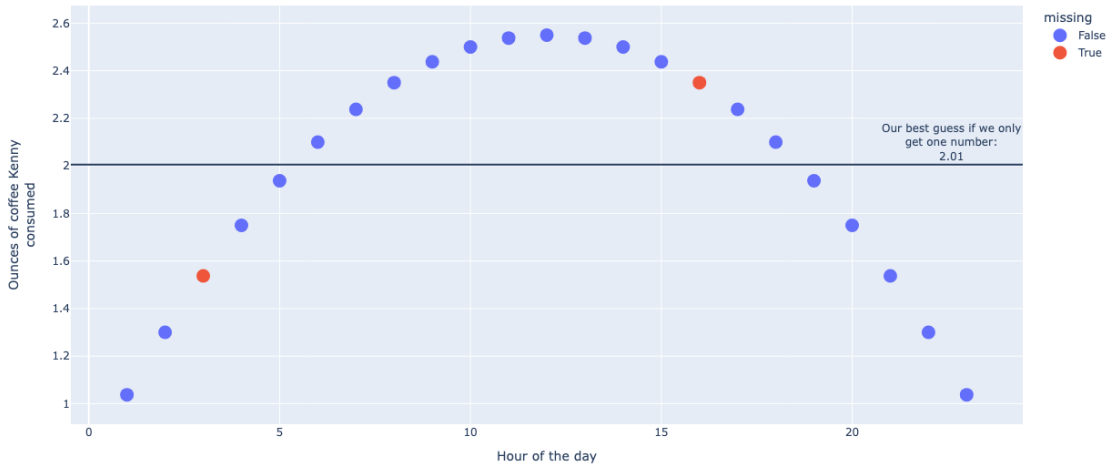
3.1.2 "Predict Only One" When you only have a sample

The above tells us what to do when we know $\mathbb{E}[Y]$. Sadly, to know that, we would need all of the data from the population. What to do? Well, our friend the *law of large numbers* tells us that if we have iid samples $y_1, ...y_n$ from the population, then our sample mean—let's call that $\hat{\mu}$, converges to $\mathbb{E}[Y]$! So even those we don't *know* $\mathbb{E}[Y]$, we have something that converges to it as n increases.

So, our "predict only one" *optimal answer* is the sample mean. Let's see what that looks like with our data from above!

[4]:
```
# Compute sample mean (ignoring the "missing data")
sample_mean = data[~data.missing].y.mean()


# Plot the sample mean along with the figure
fig.add_hline(y=sample_mean,
    annotation_text=f"Our best guess if we only<br>get one number:
 ↪<br>{sample_mean:.2f}")
fig.show()
```



3.2 How good of a guess is that?

Exercise (don't peek!): How would *you* evaluate how good of a guess this is for the two missing data points?

We'll return to how I would do it in a bit.

4 The Regression Function

In the meantime, let's look at the more interesting case. To this point, we've totally ignored the information that we have on the x-axis; i.e., what attempt number it is for me. Clearly, there is some valuable information there!

Put another way, if instead of just making a prediction based on our samples $y_1, ...y_n$, we *also* used information from our $x_1, ..., x_n$s, we should be able to make better predictions.

Put yet another way, then, we want our predictions to *be a function of the Xs!*. Let's call that function $f(X)$.

Now we can again ask, what should the function be to minimize our MSE? In other words, what should we set f as to minimize $MSE(f)$?

In the notation from above, we want to solve:

$$h^* = \underset{f}{\operatorname{argmin}}\, MSE(f(X)) = \underset{f}{\operatorname{argmin}}\, \mathbb{E}\left[(Y - f(X))^2\right]$$

With similar math as above (see Shalizi, 1.12-1.15 if you want to do it out yourself), we get that:

$$h^* = \mathbb{E}[Y|X = x]$$

That is, the optimal prediction in terms of MSE for Y when we have information about X is to take the *conditional expected value* at that value of x. This equation above is called the **regression function**.

5 Now what?

Let's come back to our machine learning perspective, now. When we say we want to define a model/hypothesis class, we can now say precisely what we are doing:

- We are building a model/hypothesis class *for the regression function.*
- The regression function, in turn, makes an assumption that our *loss function is MSE*

What model class might we choose that optimizes for our goals above—parsimony, efficient training, efficient prediction, and model expressivity? The most common answer to this question for a very long time in both statistics and machine learning has been a **linear model**. In a **linear model**, we *choose* to approximate the regression function with a function that is linear in the model parameters.

In the case where x is a single variable, then, we approximate the regression function with the function $b_0 + b_1 x$.

We will sometimes write these parameters as a vector $\beta = [b_0, b_1, \ldots, b_p]^\top$ for compact notation — you'll see this in the linear algebra section below.

Exercise: Why don't we just use $b_1 x$ as the approximation?

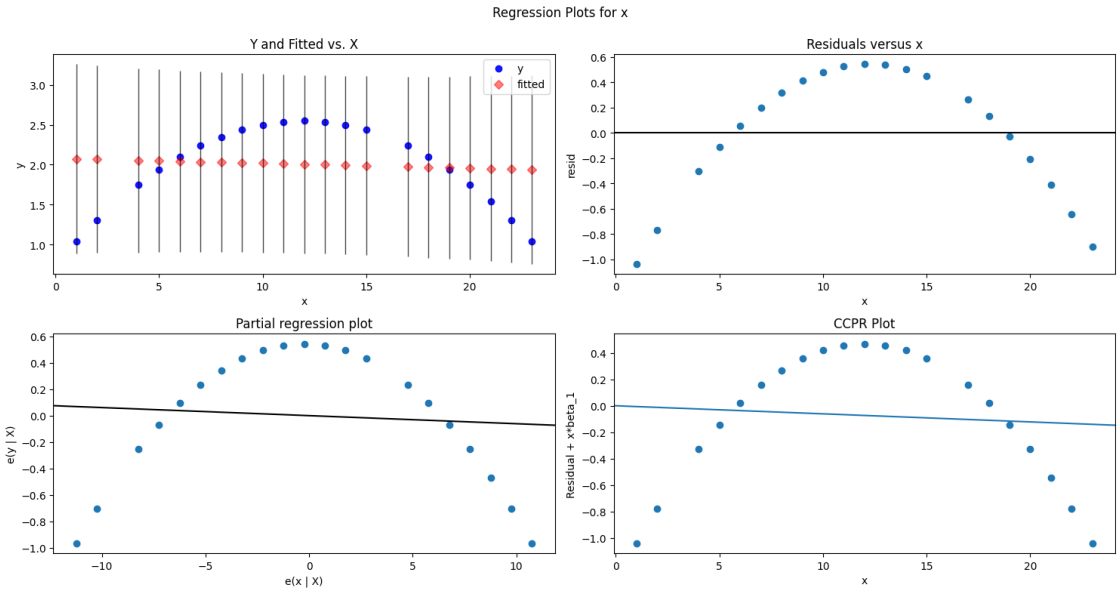
6 Returning to our simple example

Neat! Let's run a linear regression

[7]:
```
import statsmodels.api as sm
from statsmodels.formula.api import ols

f = 'y~x'
model = ols(formula=f, data=data[~data.missing]).fit()

fig = plt.figure(figsize =(15,8))
fig = sm.graphics.plot_regress_exog(model, 'x', fig=fig)
```



6.1 Making and evaluating predictions

[8]:
```
model.predict(data[data.missing])
```

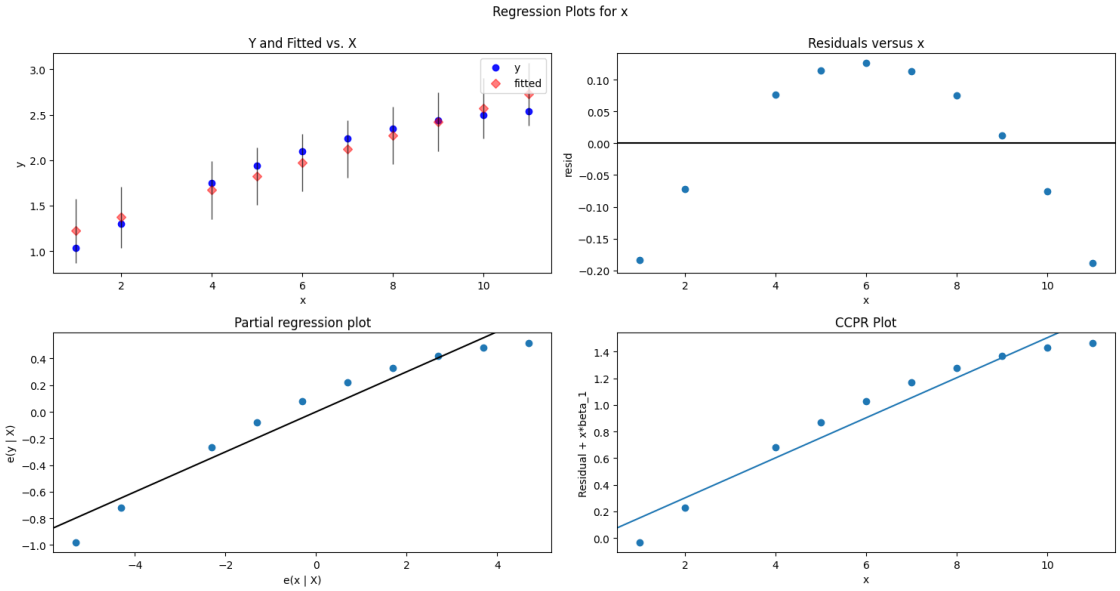
[8]:
```
2     2.061862
15    1.982348
dtype: float64
```

Uh. Not so good. What if we make the task a little less trivial?

[9]:
```
import statsmodels.api as sm
from statsmodels.formula.api import ols

f = 'y~x'
model = ols(formula=f, data=data[(~data.missing)&(data.x < 12)]).fit()

fig = plt.figure(figsize =(15,8))
fig = sm.graphics.plot_regress_exog(model, 'x', fig=fig)
```

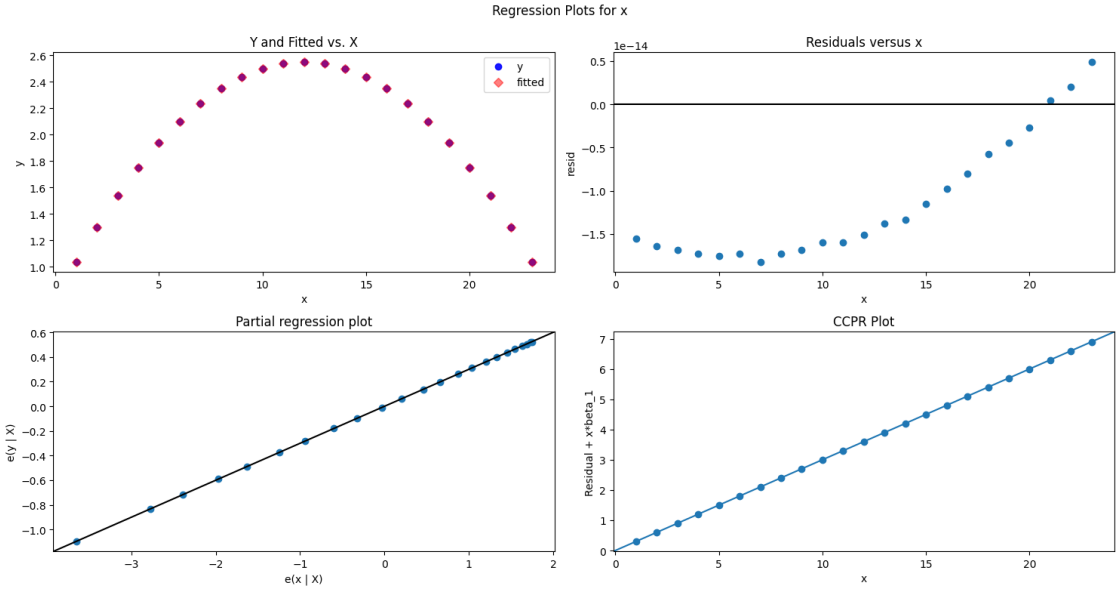


[10]:
```
model.predict(data[data.x == 3]), data[data.x == 3].y
```

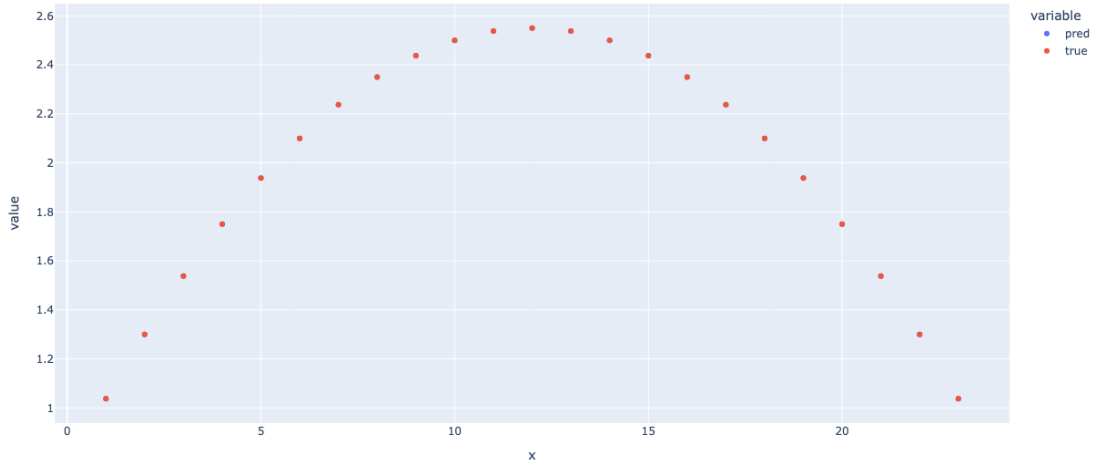
[10]:
```
(1    1.371978
 dtype: float64,
 1    1.3
 Name: y, dtype: float64)
```

[12]:
```
data['x_sq'] = data['x'].pow(2)
f = 'y~x+x_sq'
model = ols(formula=f, data=data).fit()

fig = plt.figure(figsize =(15,8))
fig = sm.graphics.plot_regress_exog(model, 'x', fig=fig)
```



[13]:
```
results = pd.DataFrame(zip(model.predict(data),data.y,data.x),␣
 ↪columns=['pred','true','x'])
px.scatter(pd.melt(results,id_vars="x"),
           x="x",y="value",color="variable",
          height=600,width=700)
```



7 Linear Regression via Linear Algebra

So far we have described linear regression conceptually and used libraries to fit models. But how does it actually work under the hood? The answer is elegant: we can express and solve linear regression entirely with linear algebra.

7.1 Setting up the problem

Suppose we have n observations and p features. We want to fit a model:

$$\hat{y} = b_0 + b_1 x_1 + b_2 x_2 + \cdots + b_p x_p$$

We can write all n predictions at once as a matrix-vector product. First, build the **design matrix** $\mathbf{X}$ by prepending a column of 1s (this handles the intercept b_0 automatically):

$$\mathbf{X} = \begin{bmatrix} 1 & x_1^{(1)} & \cdots & x_p^{(1)} \\ 1 & x_1^{(2)} & \cdots & x_p^{(2)} \\ \vdots & \vdots & & \vdots \\ 1 & x_1^{(n)} & \cdots & x_p^{(n)} \end{bmatrix}, \quad \beta = \begin{bmatrix} b_0 \\ b_1 \\ \vdots \\ b_p \end{bmatrix}, \quad \mathbf{y} = \begin{bmatrix} y^{(1)} \\ y^{(2)} \\ \vdots \\ y^{(n)} \end{bmatrix}$$

Then all predictions are simply $\hat{\mathbf{y}} = \mathbf{X}\beta$.

7.2 The loss function in matrix form

Our MSE loss over all n points is:

$$\mathcal{L}(\beta) = \frac{1}{n}\|\mathbf{y} - \mathbf{X}\beta\|^2 = \frac{1}{n}(\mathbf{y} - \mathbf{X}\beta)^\top(\mathbf{y} - \mathbf{X}\beta)$$

7.3 Solving for the optimal β: The Normal Equations

To minimize, take the gradient with respect to β and set it to zero:

$$\nabla_\beta \mathcal{L} = -\frac{2}{n}\mathbf{X}^\top(\mathbf{y} - \mathbf{X}\beta) = \mathbf{0}$$

… the details of the rest of this are not for this class :)

8 Great! But...

We've already established that linear models aren't very good. What can we do instead?

Essentially, we can replace our approximation of $\mathbb{E}[Y|X = x]$ with something that is better than a linear model. That is, we can **select a different model class**.

What, exactly?

Well, there are many options. But one model class that is very popular these days are neural networks. This is where we'll head in this class. We'll talk about neural networks generally at first, and then more specifically about the transformer architecture

9 Predicting with a Neural Network

Linear regression has a clean closed-form solution. But what if the true relationship is nonlinear? A **neural network** is a much more flexible model class that can learn arbitrary nonlinear functions — at the cost of no closed-form solution and more data/tuning required.

9.1 What is a neural network (briefly, to be extended in slides next class)?

A neural network stacks **layers** of linear transformations interleaved with **nonlinear activation functions**:

$$\mathbf{l}^{(1)} = \sigma\big(\mathbf{W}^{(1)}\mathbf{x} + \mathbf{b}^{(1)}\big)$$
$$\mathbf{l}^{(2)} = \sigma\big(\mathbf{W}^{(2)}\mathbf{l}^{(1)} + \mathbf{b}^{(2)}\big)$$
$$\hat{y} = \mathbf{W}^{(3)}\mathbf{l}^{(2)} + b^{(3)}$$

where σ is an activation function such as ReLU ($\max(0, z)$) or tanh, and $b^{(3)}$ is a scalar (not a vector) because the output $\hat{y}$ is a single real number. Without the activations, stacking linear layers would still just give a linear model — the nonlinearities are what give neural networks their expressive power.

Training uses **gradient descent**: we compute the gradient of the MSE loss with respect to all weights $\mathbf{W}$ and biases $\mathbf{b}$ (via backpropagation) and take small steps downhill.

9.2 Fitting a neural network to the coffee toy example

Let's use scikit-learn's `MLPRegressor` (Multi-Layer Perceptron) to fit the same toy dataset and compare to OLS.

[15]:
```
from sklearn.neural_network import MLPRegressor
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import mean_squared_error

#   Recompute OLS beta_hat (so this cell works standalone)
X_vals = data['x'].values
y_vals = data['y'].values
X_mat  = np.column_stack([np.ones(len(X_vals)), X_vals])
beta_hat, _, _, _ = np.linalg.lstsq(X_mat, y_vals, rcond=None)

#   Prepare training data (observed points only)
train_data     = data[~data.missing]
X_train_nn     = train_data[['x']].values   # shape (n_train, 1)
y_train_nn     = train_data['y'].values

# Neural networks are sensitive to input scale - standardize x
scaler         = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train_nn)

#   Define and train the network
```

```
# Architecture: 1 input → 20 ReLU neurons → 10 ReLU neurons → 1 output
nn_model = MLPRegressor(
    hidden_layer_sizes=(20, 10),
    activation='relu',
    solver='adam',
    max_iter=5000,
    random_state=42,
    learning_rate_init=0.01,
)
nn_model.fit(X_train_scaled, y_train_nn)

#   Generate predictions
x_range        = np.linspace(1, 23, 300).reshape(-1, 1)
x_range_scaled = scaler.transform(x_range)

y_nn_pred  = nn_model.predict(x_range_scaled)
y_ols_pred = np.column_stack([np.ones(len(x_range)), x_range]) @ beta_hat

# Training MSE comparison
y_train_nn_pred  = nn_model.predict(X_train_scaled)
y_train_ols_pred = np.column_stack([np.ones(len(X_train_nn)), X_train_nn]) @␣
 ↪beta_hat

print(f"Training MSE  - OLS:            {mean_squared_error(y_train_nn,␣
 ↪y_train_ols_pred):.6f}")
print(f"Training MSE  - Neural Network: {mean_squared_error(y_train_nn,␣
 ↪y_train_nn_pred):.6f}")

# Predictions for the two missing points
missing_pts    = data[data.missing][['x']].values
missing_scaled = scaler.transform(missing_pts)
nn_missing     = nn_model.predict(missing_scaled)
true_missing   = data[data.missing]['y'].values
print(f"\nPredictions for missing points (x=3, x=16):")
for x_val, nn_p, true_p in zip(missing_pts.flatten(), nn_missing, true_missing):
    ols_p = float(np.array([1, x_val]) @ beta_hat)
    print(f"  x={x_val:.0f}  true={true_p:.4f}  NN={nn_p:.4f}  OLS={ols_p:.4f}")

#   Plot
fig_nn = px.scatter(data, x='x', y='y', color='missing',
                    labels={"x": "Hour of the day",
                            "y": "Ounces of coffee Kenny consumed"},
                    title="OLS vs. Neural Network on toy data",
                    height=550, width=800)
fig_nn.update_traces(marker=dict(size=12))
fig_nn.add_scatter(x=x_range.flatten(), y=y_ols_pred, mode='lines',
                   name='OLS (linear)', line=dict(dash='dash', color='blue'))
```

```
fig_nn.add_scatter(x=x_range.flatten(), y=y_nn_pred, mode='lines',
                   name='Neural Network', line=dict(color='red'))
fig_nn.show()
```

```
Training MSE  - OLS:            0.247522
Training MSE  - Neural Network: 0.009611

Predictions for missing points (x=3, x=16):
  x=3  true=1.5375  NN=1.5128  OLS=2.0000
  x=16  true=2.3500  NN=2.3399  OLS=2.0000
```