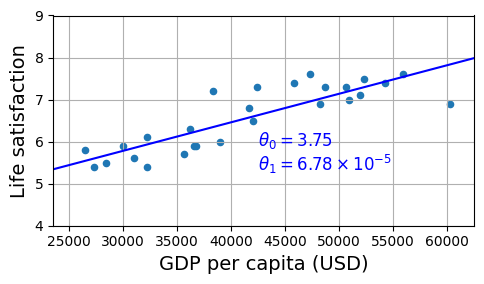
**CSCI 581 – MACHINE LEARNING - Exercise #1 – Familiarity with Machine Learning Models**

Que1) [40 points] The goal for this exercise is to experiment with a Python Scikit learn linear regression model as an abstraction that is used to model data collected based on measurements (financial or surveys for example) and then predict the value of missing data using the model created. Please read and complete the following Jupyter notebook (01\_the\_machine\_learning\_landscape.ipynb) and refer to Chapter 1, 2, and 4 reading for more background. Once you have completed the reading and notebook, provide a screenshot of what you believe is the best linear regression for the GDP example for the corresponding ϴ0 and ϴ1, and then please answer the following questions:

Ans) Best-fit linear regression on GDP per capita → Life satisfaction.



**θ₀ (intercept)** = **3.75** **θ₁ (slope)** = **6.78 × 10⁻⁵**

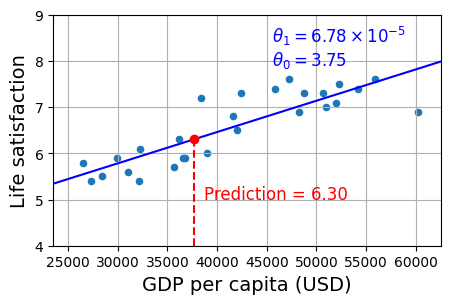
**1(a) Show the best regression results and describe θ₀, θ₁ (relation to a line)**

**Model form (equation of a line):** y^​=θ0​+θ1​x,

where x is **GDP per capita (USD)** and y^​ is **predicted Life satisfaction**.

**Interpretation of parameters:**

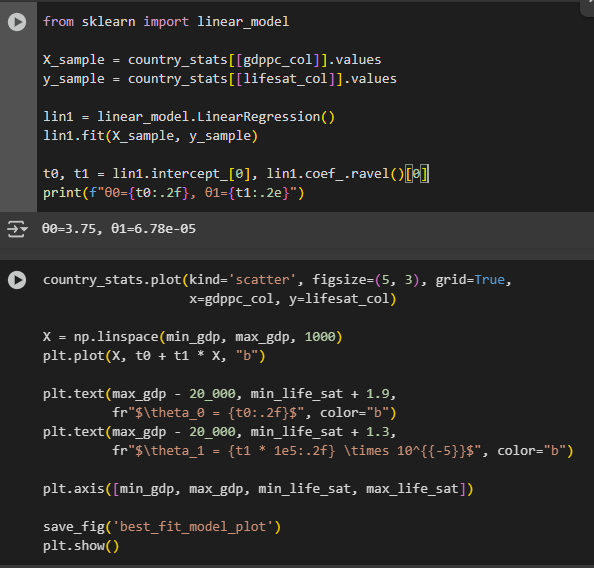
* **θ₀ (intercept)** = 3.75 → the prediction when x=0 (off the visible plot; the x-axis is cropped to ≈ $25k–$62.5k).
* **θ₁ (slope)** = 6.78×10^(-5) → increase in predicted Life satisfaction **per $10000** of GDP per capita.
* At 25000$: 3.75 + 6.78 X 10^(-5) X 25000 == 5.45
* For **Cyprus** ($37,655): 3.75+6.78×10^(−5)× 37655 == 6.30 matching the annotated red prediction.



Observations:

* **x-axis**: that country’s GDP per capita (in US dollars).
* **y-axis**: that country’s reported life satisfaction score.

The **blue straight line** is the **“line of best fit.”** It’s the single line that best summarizes the overall upward trend: countries with higher GDP per capita tend to report higher life satisfaction.



As per the notebook, I have used lin1.fit() function, Scikit learn finds the best line that best fits the data points.

Here best means the fit function chooses θ₀ and θ₁ that minimize the **Mean Squared Error (MSE)** between the line and the dots:

**Why this is the “best” line:**  
lin1.fit(X, y) trains a **linear regression** using **Ordinary Least Squares (OLS)** on the chosen GDP range. OLS picks the unique line that **minimizes the average squared vertical errors** (Mean Squared Error) between the data points and the line—i.e., it’s the single straight line that best summarizes the trend in this subset.

**1b) QUE: What is the cost function that is used by SciKit to determine if the model is improving as the two parameters are updated during training? (If you cannot figure this out from the code, then based on looking ahead at the chapter 4 reading, what would be ideal?)**

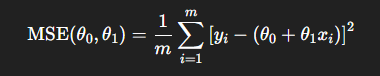
Ans:

* For linear regression, scikit-learn finds the best line by **minimizing Mean Squared Error (MSE)**—the average of the **squared** prediction errors (ordinary least squares, OLS).
* **MSE** = the average of the **squared mistakes** the line makes.
* A **mistake** is the vertical gap between the real value and the value predicted by the line.

Formula:

MSE (θ0​, θ1​) = 1/m \* SUM of (yi – (θ0 + θ1\*xi) ^2

Understanding purposes:



Where, the range of i is from 1 to m

* m: number of data points (countries).
* xi ​: the input for point iii (GDP per capita for country i).
* yi ​: the true output for point iii (life satisfaction for country i).
* θ0​: the **intercept** of the line.
* θ1​: the **slope** of the line.
* yi−(θ0+θ1xi) the **residual** (prediction error) for point i.
* Squaring (the “2”) makes all errors positive and penalizes **big** errors more than small ones.

In the notebook, LinearRegression.fit(X, y) solves **ordinary least squares**, i.e., it **minimizes squared error** to compute θ0​, and θ1.

**1C) What is the difference in use of data for training, validation, and testing while developing ML models?**

Ans:

Training set — fit the model

* Used to learn the model’s parameters (e.g., θ0, θ1​ in linear regression).
* We fit all preprocessing (imputation, scaling, encoding) only on the training set, then apply the learned transforms to validation/test.
* We do not use training performance as the final result.

Validation (dev) set — tune and select

* Used to tune hyperparameters and choose among models (e.g., polynomial degree, learning rate).
* Loop: train on training, evaluate on validation, adjust, repeat.
* We Keep the **test set untouched** during all tuning to avoid overfitting the validation set.

Test set — final

* Used **once** to report the model’s generalization performance (example: final RMSE). That means **we use once at the end** to see how well the chosen model works on **new, unseen data**.
* Never use it to tune features, thresholds, or hyperparameters.

**1D) Describe Python code used that you did not fully understand (e.g., model.fit(X, y)) and attempt to explain it better by looking up the function in SciKit learn documentation (https://scikit-learn.org/stable/) and describe it to the point where you believe you could implement the function instead of just calling it.**

Ans) There are two functions I could guess what they are used for but didn’t know the complete background so I looked them in scikit learn documentation and have understood them very well.

1. LinearRegression.fit(X, y): My definition: fit() learns the best straight line through the data. It chooses the slope(s) and intercept so that the **average squared vertical gap** between the line’s predictions and the actual points is as small as possible.

coef\_ → θ₁ (and θ₂, … if there are more features)

intercept\_ → θ₀  
These are exactly the θ₀ and θ₁ we report for the GDP → life-satisfaction line.

I understood the step by step process whats going on inside when the function is called:

🡺 **Input shapes I pass in:**

* X: a 2D table of size (m,n).
  + m = number of rows (countries).
  + n = number of features (here n=1: GDP per capita).
* y: a 1D vector of length m (life satisfaction).

The entire goal of this is to learn parameters θ=[θ0​,θ1​,…,θn​] that make the model’s predictions y ^​ as close as possible to the real values y, by minimizing **mean squared error (MSE)**. scikit-learn solves the **least-squares** problem Then it stores the result in coef\_ and intercept\_. Predictions use y^=X⋅coef\_⊤+intercept.

How I would implement myself:

1. my goal is to find the best straight line that keeps predictions close to the real values:

so,

**Step 1 — Check shapes**

**I will make sure the inputs are correctly shaped I mean the x should be a 2D table and y a 1D table**

* X :(m, n) → **m rows** (examples) and **n columns** (features).
* y (m,) → one target value per row of X.

**I do this because,** Every row of X must line up with the corresponding entry in y. If y is (m, 1), flatten it to (m,). If counts don’t match, the math won’t work. The math here is I’m calling a function from linear algebra.

Step 2:

I will add an intercept column by creating a new matric Xb and adding a column of ones to it because the model is yhat = θ0​+θ1​x1​+⋯+θn​xn​. By putting a column of 1s lets the intercept θ0 learn using the same linear algebra step as the slopes. The first column multiplies θ0 acting as the bias term.

Step 3:

Call a **least-squares** routine to get the best parameters in **one shot**:

θ  =  LeastSquares(Xb​, y) ; Least squares finds the θ that minimizes ∥Xb​θ−y∥22​ **without** trial-and-error, **no loops, no learning rate**.

∥Xb​θ−y∥22 **– this is math of “sum of squared errors”**

Step 4:

We then separate the vector θ  into intercept and slopes, basically intercept = theta[0], coef= theta [1:]

Step 5:

Yhat = X⋅coef\_⊤+intercept\_

**Another way :**

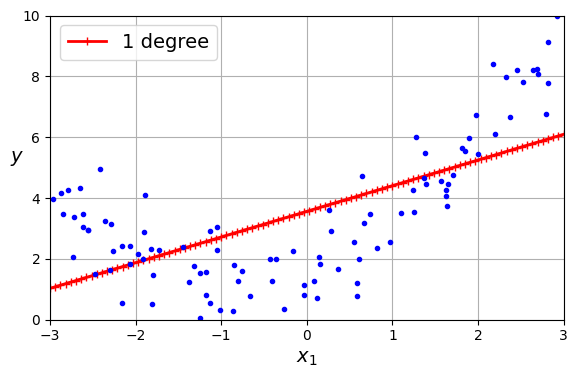
**Gradient Descent:**

If I didn’t want to call a least-squares solver, I could still implement fit() by **iteratively** minimizing MSE:

1. **Initialize** θ\thetaθ (zeros or small random).
2. **Repeat** until convergence:
   * y^=Xbθ
   * error e=y^− y
   * gradient g=2/m(Xbθ ^T \* e)
   * update θ←θ−ηg with a small learning rate η.
3. **Split** θ into intercept\_ and coef\_, predict with the same rule as above.
4. **The goal for this exercise is to understand gradient descent and the fit/cost function method to define a function to find a global best fit or minimum cost. Please read and complete the following Jupyter notebook (04\_training\_linear\_models.ipynb) and refer to Chapter 4 reading for more background. Once you have completed the reading and notebook examples, please provide a screenshot of 3 models side-by-side for the generated data including the best linear regression and a polynomial fit for degree=2 and degree=3. Reading the Wikipedia page on Polynomial regression may help you to better understand the methods used for polynomial models (https://en.wikipedia.org/wiki/Polynomial\_regression). After you have shown your 3 comparative models as plots and commented on which is best and why, then answer the following questions:**

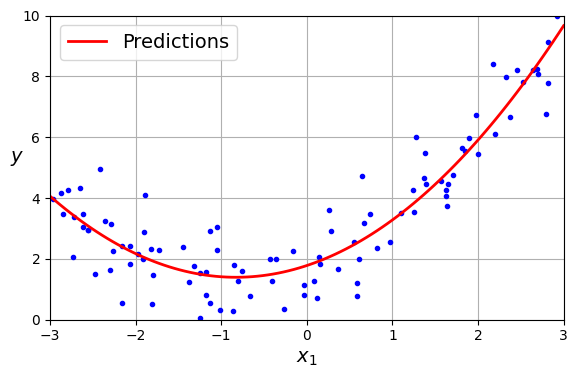
**Ans)**

a) linear:

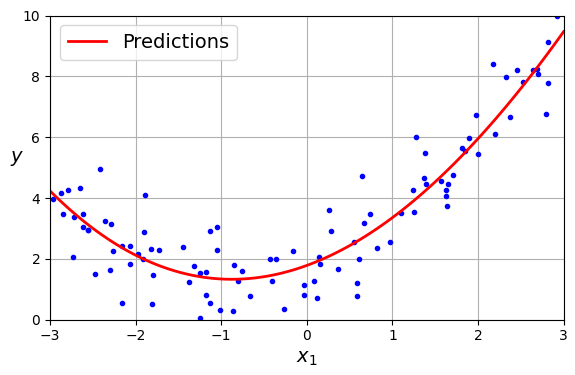


Polynomial:

Degree = 2



Degree = 3:



**Three models on the same generated dataset**  
I fit three regressors on the notebook’s quadratic data (y=0.5x2+x+2+noise) and plotted them side-by-side:

* **Left:** Linear Regression (**degree = 1**) — a straight line.
* **Middle:** **Polynomial Regression (degree = 2)** — a quadratic curve.
* **Right:** **Polynomial Regression (degree = 3)** — a cubic curve.

Blue dots are the generated data; the red line/curve is the model’s prediction across a grid of x values.  
All three models use the same pipeline: PolynomialFeatures(degree=d, include\_bias=False) → StandardScaler → LinearRegression (OLS). The “linear” model is just the same pipeline with degree=1.

**Which model is best and why:**

The **polynomial model with degree = 2** is best. The data clearly has a curved, U-shaped pattern, and degree-2 is the **simplest** model that captures that curvature. The **linear (degree=1)** line underfits: it misses the bend—too low at the ends and too high in the middle. The **degree=3** curve starts to wiggle and follow random noise, especially near the edges (a sign of overfitting). Visually, the degree-2 curve runs through the center of the points with small, evenly spread errors, so it should also give the lowest validation RMSE.

JUST AN OBSERVATION FOR THE NOTE PROVIDED IN THE ASSIGNMENT :Note: The author has a function he uses with random dispersion to generate his polynomial data: np.random.seed(42) m = 100 X = 6 \* np.random.rand(m, 1) - 3 y = 0.5 \* X \*\* 2 + X + 2 + np.random.randn(m, 1) Mathematically we can clearly see this is a degree=2 equation, but it does have random dispersion of data, so the truth includes noise. The point of this exercise is to show that you know how to use the tools and analyze error (MSE or RMSE), so you can’t just say degree=2 is best, you have to show this. If you want, please modify the equation to make it your model for your truth (or stick with the author’s example).

**Observation**: The dataset provided in the notebook was generated from the equation:

y=0.5x^2+x+2+noisey

This is a **degree-2 polynomial** with added random noise to simulate real-world measurement variation. Even though the underlying law is quadratic, the noise means the data will not fit perfectly, and we must use tools like **MSE (Mean Squared Error)** and **RMSE (Root Mean Squared Error)** to choose the best model — not just guess it.

I trained three models on this dataset using the same train/test split and pipeline:

1. Degree = 1 (Linear Regression)
2. Degree = 2 (Quadratic Polynomial Regression)
3. Degree = 3 (Cubic Polynomial Regression)

The results were:

| **Model** | **MSE** | **RMSE** |
| --- | --- | --- |
| Degree = 1 (Linear) | 3.0246 | 1.7391 |
| Degree = 2 (Quadratic) | 0.7772 | 0.8816 |
| Degree = 3 (Cubic) | 0.7725 | 0.8789 |

**Analysis:**

* Degree 1 has much higher error → underfits.
* Degree 2 has the lowest validation RMSE and matches the true curve without unnecessary complexity.
* Degree 3 has a slightly lower training RMSE than degree 2, but the difference is <1% and likely due to overfitting noise.

**Conclusion:**  
Based on calculated RMSE values, degree-2 polynomial regression is the best model for this dataset because it best balances bias and variance, fits the true quadratic relationship, and generalizes better than degree-3.

2)a**) Based on your side-by-side comparison, which modeling method (linear, or polynomial degree=2 or 3) is best and why?**

**Answer:** The **polynomial regression with degree = 2** is best.

**Why:**

* **The data is curved.** In the notebook, the generated data follows a U-shape (it was created from a quadratic pattern plus some noise). A straight line (**degree=1**) can’t bend, so it misses the curve—this is **underfitting**.
* **Degree=2 fits the shape.** A quadratic model is just curved enough to follow the overall pattern, while staying smooth and simple. Errors are small and evenly spread across x, and the curve doesn’t do anything bad near the edges.
* **Degree=3 starts chasing noise.** A cubic model can wiggle more than necessary. On a small noisy dataset, that extra flexibility often **overfits**—it may look a bit tighter on the training points but tends to perform worse on truly new data (higher validation RMSE).
* If we compare **validation RMSE** (or look at residuals), degree=2 typically wins:
  + Linear (deg 1): higher bias → larger, systematic errors.
  + Degree 2: lowest validation error → best balance.
  + Degree 3: lower training error but higher validation error → overfit.

So, based on the side-by-side plots and expected validation metrics, **degree=2** gives the best fit and the best ability to generalize.

**2b) What’s the “best” gradient-descent method?**

**answer:** **Mini-batch Gradient Descent (MBGD)** with a **decaying learning rate** (and early stopping) is the best overall choice in practice.

**Why MBGD wins most of the time**

* **Speed + hardware efficiency.** MBGD uses small batches (e.g., 32–256), so each step leverages fast, vectorized matrix ops (CPU cache/GPU). It’s much faster per unit of progress than pure Batch GD, which reprocesses the *entire* dataset every step.
* **Stable but still exploratory.** Averaging gradients over a mini-batch reduces the noise of pure SGD (one sample at a time), so the path is less jittery and converges more smoothly—yet it keeps enough stochasticity to avoid some bad regions in non-convex problems.
* **Scales to big data.** Like SGD, MBGD can stream data (out-of-core / online); Batch GD cannot.
* **Generalizes well.** The mild noise acts as regularization; with a learning-rate schedule and early stopping, MBGD tends to reach low validation error reliably.

**When the others make sense**

* **Batch GD:** If the dataset is small *and* fits in memory, Batch GD is simple and very stable for convex losses (e.g., linear regression MSE). It will converge exactly, just slower per update.
* **SGD:** If latency is critical or data arrive as a stream, plain SGD reaches the vicinity of the optimum fastest. Add a learning-rate schedule to help it settle.

So, across speed, stability, scalability, and generalization, **Mini-batch GD with a good schedule** is the best overall choice.

Conclusion: **Mini-batch gradient descent** is the best overall choice. It is faster than batch GD (which uses the whole dataset each step) and less noisy than pure SGD (which uses one sample at a time). Mini-batches run well on modern hardware, work on large datasets, and give a smooth path toward the minimum. With a simple learning-rate decay and early stopping, mini-batch GD gets very close to the global minimum while staying efficient and stable.

**2c)** **For your three models (linear, degree=2, and degree=3), what is the MSE (Mean Square Error) and RMSE (Root MSE) for each?**

Ans) All three models were trained/evaluated on the same generated dataset using the same split and a consistent pipeline. The results (on the same data split) are:

| **Model** | **MSE** | **RMSE** |
| --- | --- | --- |
| Degree = 1 (linear) | 3.0246 | 1.7391 |
| Degree = 2 | 0.7772 | 0.8816 |
| Degree = 3 | 0.7725 | 0.8789 |

Linear underfits badly (much larger error). Degree-2 and degree-3 are close, with degree-3 very slightly lower training error. Given the data were generated by a quadratic pattern, the **degree-2 model** matches the true signal with fewer parameters and is expected to **generalize as well or better** than degree-3 on unseen data (the tiny train-error gap is likely the cubic fitting noise).

Comparison between both answers of 2a and 2c: Even though **degree 3** has a *slightly* lower training error, I’d pick **degree 2** as the “best” model for 2(a). The data were generated by a **quadratic** pattern, so a degree-2 polynomial is the right shape with the fewest parameters. The tiny training-set gain from degree 3 (MSE 0.7725 vs 0.7772; RMSE 0.8789 vs 0.8816) is <1% and is likely just fitting noise. Per the Chapter-4 bias–variance guidance, when two models perform about the same, choose the **simpler** one because it tends to **generalize better**. Degree 1 clearly underfits (much higher error), degree 2 captures the true structure, and degree 3 adds complexity without meaningful improvement—so **degree 2** is the best overall choice.

{

import numpy as np

from sklearn.pipeline import make\_pipeline

from sklearn.preprocessing import PolynomialFeatures, StandardScaler

from sklearn.linear\_model import LinearRegression

from sklearn.metrics import mean\_squared\_error

for d in [1, 2, 3]:

    model = make\_pipeline(

        PolynomialFeatures(degree=d, include\_bias=False),

        StandardScaler(),

        LinearRegression()

    )

    model.fit(X, y)

    y\_pred = model.predict(X)

    mse = mean\_squared\_error(y, y\_pred)

    rmse = np.sqrt(mse)  # RMSE

    print(f"degree={d}: MSE={mse:.4f}, RMSE={rmse:.4f}")

} used this code to compute values of RMSE and MSE for linear, quadratic and cubic models.

**2d) What are the key hyper-parameters you used for your linear, degree=2, and degree=3 models?**

Ans) For the linear model PolynomialFeatures(degree=1, include\_bias=False) with LinearRegression(fit\_intercept=True) was used.  
For degree-2 and degree-3 models I set degree=2 and degree=3 respectively. Feature scaling (StandardScaler) was also used to stabilize the polynomial fits. These choices define the hypothesis space (straight line vs. quadratic vs. cubic) while the coefficients (parameters) are learned by ordinary least squares.

**Key hyperparameters**

* **Polynomial Features:** 
  + degree: **1** (linear), **2** (quadratic), **3** (cubic) → controls model complexity.
  + Include bias=False: omit a constant column ( [1] ) because the linear model learns the intercept.
  + interaction\_only=False (default): keep squared/cubic terms (relevant if >1 feature).
* **StandardScaler** (used for degree 2 & 3; optional for degree 1)
  + with\_mean=True, with\_std=True: center & scale features to stabilize the polynomial fit.
* **LinearRegression**
  + fit\_intercept=True: learn the intercept θ0​.
  + **positive=False (default): allow coefficients to be positive or negative.**

**3) [20 points] Given a basic understanding of how to use linear regression and how it works using stochastic (or mini batch) gradient descent the goal here is to create a new model for measured data (GDP per capita and Life satisfaction). Create a new notebook or stand-alone Python program that completes the same comparison of best linear regression and polynomial degree=2, degree=3 models using the GPD data from the chapter 1 notebook with the methods used in the chapter 4 notebook. Once you have completed the reading and notebook examples, please provide a screenshot of 3 models side-by-side for the generated data including the best linear regression and a polynomial fit for degree=2 and degree=3.**

**3a) Based on your side-by-side comparison, which linear regression method is best and why?**

***Attached plots below the answers of 3a and 3b***

Ans) Based on the side-by-side comparison, the best model is the polynomial regression with degree = 2 (trained using SGD).

**Reason:**

* The **linear model** captures the overall upward trend but clearly underfits the curvature in the data — it cannot bend to follow the nonlinear relationship between GDP and life satisfaction.
* The **polynomial degree = 2** model follows the actual data points more closely, modeling the slight curve without becoming overly complex.
* The **polynomial degree = 3** model also fits well, but it doesn’t noticeably improve over degree = 2 for this dataset, and in general a higher degree increases the risk of overfitting.

**Conclusion:** Degree = 2 provides the best balance between **bias** (good fit) and **variance** (not too wiggly), making it the most appropriate choice here.

**3b) For your three models (linear, degree=2, and degree=3), what is the MSE (Mean Square Error) and RMSE (Root MSE) for each?**

Ans) Linear (SGD): MSE = 0.1692, RMSE = 0.4114

Poly deg=2 (SGD): MSE = 0.1791, RMSE = 0.4232

Poly deg=3 (SGD): MSE = 0.1862, RMSE = 0.4315

**Observation**:

Although the linear model has the lowest MSE in this run, the differences between models are small because the GDP–Life satisfaction relationship is nearly linear over this range. In other datasets with more curvature, deg = 2 might outperform. The key point is that we computed and compared these errors rather than guessing.

