Régression Linéaire Python

Aldj Souleïman
Buccafurri Arnaud
Gaillot Solène
M1 Économétrie-Statistiques - Paris 1 Panthéon-Sorbonne

Novembre 2025

Résumé

Dans ce projet, nous avons construit un programme Python qui génère une régression linéaire. Ce document s'articule autour de trois parties : une première avec notre code qui génère la régression, une seconde où l'on test les différentes fonction puis une troisième qui présente trois exemples d'applications.

Table des matières

1 Code

```
import numpy as np
import pandas as pd
from typing import List, Tuple, Dict



class DataSet:
    """
    Stores a feature matrix (X), a target vector (y), and their column names.
    Adds an intercept to the feature matrix and converts the data into
    pandas DataFrames with a train/test split.
    """

    def __init__(
        self,
        X: np.ndarray,
        y: np.ndarray,
        features_name: List[str],
        target_name: List[str]
    ) -> None:
        """
        Initializes the DataSet instance and reshapes the target vector
        into a column vector.

        Parameters:
            X : np.ndarray.
            y : np.ndarray.
            features_name : List[str].
            target_name : List[str].

        Returns:
            None.
        """

        self.X = X
        self.y = y.reshape(-1, 1)
        self.features_name = features_name
        self.target_name = target_name

    def add_intercept(self) -> None:
        """
        Raises an error if there is a dimension mismatch between X and y
        Adds a column of ones as the first column of the feature matrix.
        Also adds the name "intercept" at the beginning of the feature names list.

        Returns:
            None.
        """
        if self.X.shape[0] != self.y.shape[0]:
            raise ValueError("X and y must have the same number of rows")
        intercept = np.ones((self.X.shape[0], 1))
        self.X = np.concatenate((intercept, self.X), axis=1)
        self.features_name = ["intercept"] + self.features_name

    def turn_into_dataframe(
        self,
    ) -> Tuple[pd.DataFrame, pd.DataFrame, pd.DataFrame, pd.DataFrame]:
        """
        Converts the feature matrix (X) and the target vector (y)
        into four pandas DataFrames using an 80/20 train test split.

        Returns:
            X_train : pd.DataFrame.
            y_train : pd.DataFrame.
            X_test : pd.DataFrame.
            y_test : pd.DataFrame.
        """

        df_X = pd.DataFrame(self.X, columns=self.features_name)
        df_y = pd.DataFrame(self.y, columns=self.target_name)

        cutoff = int(len(self.X) * 0.8)
        X_train = df_X.iloc[:cutoff]
        y_train = df_y.iloc[:cutoff]
        X_test = df_X.iloc[cutoff:]
        y_test = df_y.iloc[cutoff:]

        return X_train, y_train, X_test, y_test


class LinearRegression:
    """
    Stores the feature names and model coefficients, and implements
    Ordinary Least Squares (OLS) regression using the closed-form
    normal equation. Includes methods for fitting the model on training
    data and predicting target values for unseen samples (test data).
    """

    def __init__(self, features_name: List[str]) -> None:
        """
        Initializes the LinearRegression instance.

        Parameters:
            features_name : List[str].

        Returns:
            None.
        """
        self.coeficients = None
        self.features_name = features_name

    def fit_model(
        self,
        X_train: pd.DataFrame,
        y_train: pd.DataFrame
    ) -> np.ndarray:
        """
        Fits the OLS model on the training data
        and returns predictions for the training set.
        Raises an error if the features matrix is not invertible.

        Parameters:
            X_train : pd.DataFrame.
            y_train : pd.DataFrame.

        Returns:
            y_pred : np.ndarray.
        """
        X = X_train.values
        y = y_train.values
        X_t = X.T

        try:
            self.coeficients = np.linalg.inv(X_t @ X) @ X_t @ y
        except np.linalg.LinAlgError as error:
            raise np.linalg.LinAlgError(
                "Matrix X cannot be inverted."
                "Check for multicollinearity or duplicate features."
            ) from error

        y_pred = X @ self.coeficients
        return y_pred

    def predict_model(self, X_test: pd.DataFrame) -> np.ndarray:
        """
        Predicts target values for new samples (test data).
        Raises an error if predict_model is called before the model has been fitted.

        Parameters:
            X_test : pd.DataFrame.

        Returns:
            y_pred : np.ndarray.
        """

        if self.coeficients is None:
            raise ValueError("Model has not been fitted yet")

        X = X_test.values
        y_pred = X @ self.coeficients
        return y_pred

    def to_dict(self) -> Dict[str, float]:
        """
        Returns the coefficients as a dictionary mapping
        each feature name to its associated coefficient.
        Returns:
            coefficients : Dict[str, float].
        """

        coefficients = {}
        for i in range(self.coeficients.shape[0]):
            coefficients[self.features_name[i]] = float(
                round(self.coeficients[i, 0], 3)
            )
        return coefficients


class Result:
    """
    Stores a fitted linear regression model with the true and predicted
    values for both training and test sets. Computes evaluation metrics such as
    R , MSE, and RMSE for model assessment.
    """

    def __init__(
        self,
        model: LinearRegression,
        y_train: pd.DataFrame,
        y_pred_train: np.ndarray,
        y_test: pd.DataFrame,
        y_pred_test: np.ndarray
    ) -> None:
        """
        Initializes the Result instance.

        Parameters:
            model : LinearRegression.
            y_train : pd.DataFrame.
            y_pred_train : np.ndarray.
            y_test : pd.DataFrame.
            y_pred_test : np.ndarray.

        Returns:
            None.
        """

        self.model = model
        self.R2 = None
        self.y_train = y_train.values
        self.y_pred_train = y_pred_train
        self.y_test = y_test.values
        self.y_pred_test = y_pred_test

    def get_r2(self, y_true: np.ndarray, y_pred: np.ndarray) -> float:
        """
        Computes the R .
        Parameters:
            y_true : np.ndarray.
            y_pred : np.ndarray.

        Returns:
            R2 : float.
        """

        scr = float(np.sum((y_true - y_pred) ** 2))
        sct = float(np.sum((y_true - np.mean(y_true)) ** 2))
        R2 = 1 - (scr / sct)
        return R2

    def calculate_error(
        self,
        y_true: np.ndarray,
        y_pred: np.ndarray
    ) -> Tuple[float, float]:
        """
        Computes the Mean Squared Error (MSE) and Root Mean Squared Error (RMSE).

        Parameters:
            y_true : np.ndarray.
            y_pred : np.ndarray.

        Returns:
            mse : float.
            rmse : float.
        """

        mse = float(np.mean((y_true - y_pred)**2))
        rmse = float(np.sqrt(mse))
        return mse, rmse

    def results(self) -> str:
        """
        Generates a summary of the model performance, including
        coefficients, R  , MSE, and RMSE for both training and test sets.

        Returns:
            A summary string containing:
            - coefficients,
            - train R  and test R  ,
            - train MSE and RMSE,
            - test MSE and RMSE.
        """

        r2_train = self.get_r2(self.y_train, self.y_pred_train)
        r2_test = self.get_r2(self.y_test, self.y_pred_test)

        mse_train, rmse_train = self.calculate_error(
            self.y_train, self.y_pred_train
        )
        mse_test, rmse_test = self.calculate_error(
            self.y_test, self.y_pred_test
        )

        return (
            f"Model coefficients: {self.model.to_dict()}.\n"
            f"Train R  : {round(r2_train, 3)}.\n"
            f"Test R  : {round(r2_test, 3)}.\n"
            f"Train RMSE: {round(rmse_train, 3)} (MSE={round(mse_train, 3)}).\n"
            f"Test RMSE: {round(rmse_test, 3)} (MSE={round(mse_test, 3)})."
        )



def main(X: np.ndarray,
         y: np.ndarray,
         features_name: list[str],
         target_name: list[str]) -> None:
    """
    Runs the full regression on user-provided data:
    - builds the dataset object,
    - adds an intercept column,
    - converts data into train/test DataFrames,
    - fits a linear regression model,
    - generates predictions,
    - computes evaluation metrics and prints a summary.

    Parameters:
        X : numpy.ndarray of shape (n_samples, n_features).
        y : numpy.ndarray of shape (n_samples,) or (n_samples, 1).
        features_name : list of str.
        target_name : list of str.

    Returns:
        None.
    """

    data = DataSet(X, y, features_name, target_name)
    data.add_intercept()


    X_train, y_train, X_test, y_test = data.turn_into_dataframe()

    model = LinearRegression(data.features_name)
    y_pred_train = model.fit_model(X_train, y_train)
    y_pred_test = model.predict_model(X_test)

    result = Result(model, y_train, y_pred_train, y_test, y_pred_test)
    print(result.results())


from sklearn.datasets import make_regression
```

```
X, y = make_regression(
          n_samples=15000,
          n_features=3,
          random_state=42,
          noise=50
)
features_name = ["X1", "X2", "X3"]
target_name = ["y"]

main(X, y, features_name, target_name)
```

2 Tests

2.1 class DataSet

```
#Class DataSet
def test_dataset_initializes_properly():
    X = np.random.randn(5, 2)
    y = np.random.randn(5)
    features_name = ["X1", "X2"]
    target_name = ["y"]

    ds = DataSet(X, y, features_name, target_name)

    assert ds.X.shape == (5, 2)
    assert ds.y.shape == (5, 1)
    assert ds.features_name == ["X1", "X2"]
    assert ds.target_name == ["y"]


def test_raises_error_on_dimension_mismatch():
    X = np.random.randn(100, 2)
    y = np.random.randn(99)

    try:
        ds = DataSet(X, y, ["X1", "X2"], ["y"])
        ds.add_intercept()
        assert False, "ValueError was not raised"
    except ValueError as error:
        assert str(error) == "X and y must have the same number of rows"


def test_add_intercept():
    X = np.random.randn(5, 2)
    y = np.random.randn(5)

    ds = DataSet(X, y, ["X1", "X2"], ["y"])
    ds.add_intercept()

    assert ds.X.shape == (5, 3)
    assert np.all(ds.X[:, 0] == 1)
    assert ds.features_name[0] == "intercept"


def test_turn_into_dataframe():
    X = np.random.randn(10, 2)
    y = np.random.randn(10)

    ds = DataSet(X, y, ["X1", "X2"], ["y"])
    X_train, y_train, X_test, y_test = ds.turn_into_dataframe()

    assert len(X_train) == 8
    assert len(X_test) == 2
    assert len(y_train) == 8
    assert len(y_test) == 2
    assert type(X_train) == pd.DataFrame
    assert type(X_test) == pd.DataFrame
    assert type(y_train) == pd.DataFrame
    assert type(y_test) == pd.DataFrame
```

2.2 class LinearRegression

```
#Class LinearRegression
def test_linearregression_initializes_properly():
    model = LinearRegression(["X1", "X2"])

    assert model.features_name == ["X1", "X2"]
    assert model.coeficients is None


def test_fit_model_raises_error_when_matrix_not_invertible():
    X = pd.DataFrame([[1, 2], [2, 4]], columns=["X1", "X2"])
    y = pd.DataFrame([[3], [6]], columns=["y"])

    model = LinearRegression(["X1", "X2"])

    try:
        model.fit_model(X, y)
        assert False, "np.linalg.LinAlgError was not raised"
    except np.linalg.LinAlgError:
        assert True


def test_fit_model_coefficients():
    X = pd.DataFrame([[1,2], [1,3], [2,0]], columns=["X1", "X2"])
    y = pd.DataFrame([[0], [-1], [4]], columns=["y"])

    model = LinearRegression(["X1", "X2"])
    model.fit_model(X, y)

    assert model.coeficients.shape == (2, 1)
    assert type(model.coeficients) == np.ndarray
    assert round(model.coeficients[0,0],0) == 2
    assert round(model.coeficients[1,0],0) == -1


def test_fit_model_prediction():
    X = pd.DataFrame([[1,2], [1,3], [2,0]], columns=["X1", "X2"])
    y = pd.DataFrame([[0], [-1], [4]], columns=["y"])

    model = LinearRegression(["X1", "X2"])
    y_pred = model.fit_model(X, y)

    assert y_pred.shape == (3, 1)
    assert type(y_pred) == np.ndarray
    for i in range(len(y_pred)):
        assert round(y_pred[i,0], 0) == y.iloc[i,0]


def test_predict_model():
    X = pd.DataFrame([[1,2], [1,3], [2,0]], columns=["X1", "X2"])
    y = pd.DataFrame([[0], [-1], [4]], columns=["y"])

    model = LinearRegression(["X1", "X2"])
    model.fit_model(X, y)
    y_pred = model.predict_model(X)

    assert y_pred.shape == (3, 1)
    assert type(y_pred) == np.ndarray


def test_predict_before_fit_raises_error():
    X = pd.DataFrame([[1,2], [1,3], [2,0]], columns=["X1", "X2"])
    model = LinearRegression(["X1", "X2"])

    try:
        model.predict_model(X)
        assert False, "ValueError was not raised"
    except ValueError as error:
        assert str(error) == "Model has not been fitted yet"


def test_to_dict():
    X = pd.DataFrame([[1,2], [1,3], [2,0]], columns=["X1", "X2"])
    y = pd.DataFrame([[0], [-1], [4]], columns=["y"])

    model = LinearRegression(["X1", "X2"])
    model.fit_model(X, y)
    d = model.to_dict()

    assert type(d) == dict
    assert set(d.keys()) == {"X1", "X2"}
    for v in d.values():
        assert type(v) == float
    assert d["X1"] == 2
    assert d["X2"] == -1
```

2.3 class Result

```
#Class Result
def test_result_initializes_properly():
    model = LinearRegression(["X1", "X2"])
    y_train = pd.DataFrame([[1], [2], [3]])
    y_pred_train = np.array([[1], [2], [3]])
    y_test = pd.DataFrame([[4], [5]])
    y_pred_test = np.array([[4], [5]])

    result = Result(model, y_train, y_pred_train, y_test, y_pred_test)

    assert type(result.model) == LinearRegression
    assert (result.y_train == y_train.values).all()
    assert (result.y_pred_train == y_pred_train).all()
    assert (result.y_test == y_test.values).all()
    assert (result.y_pred_test == y_pred_test).all()


def test_get_r2_correct_calculation():
    y_true = np.array([1, 2, 3]).reshape(-1, 1)
    y_pred = np.array([1, 2, 3]).reshape(-1, 1)

    result = Result(None, pd.DataFrame(y_true), y_pred, pd.DataFrame(y_true), y_pred)
    r2 = result.get_r2(y_true, y_pred)

    assert r2 == 1.0


def test_get_r2_correct_value():
    X, y = make_regression(n_samples=1000, n_features=2, noise=1, random_state=42)
    X = pd.DataFrame(X, columns=["X1", "X2"])
    y = pd.DataFrame(y, columns=["y"])

    model = LinearRegression(["X1", "X2"])
    y_pred = model.fit_model(X, y)
    result = Result(model, y, y_pred, y, y_pred)
    R2 = result.get_r2(y.values, y_pred)

    assert 0 <= R2 <= 1


def test_calculate_error():
    y_true = np.array([1, 2, 3]).reshape(-1, 1)
    y_pred = np.array([1, 2, 3]).reshape(-1, 1)

    result = Result(None, pd.DataFrame(y_true), y_pred, pd.DataFrame(y_true), y_pred)
    mse, rmse = result.calculate_error(y_true, y_pred)

    assert mse == 0.0
    assert rmse == 0.0


def test_results():
```

```
X, y = make_regression(n_samples=100, n_features=2, noise=10, random_state=42)
X = pd.DataFrame(X, columns=["X1", "X2"])
y = pd.DataFrame(y, columns=["y"])

model = LinearRegression(["X1", "X2"])
y_pred = model.fit_model(X, y)
result = Result(model, y, y_pred, y, y_pred)
output = result.results()

assert type(output) == str
assert "Model coefficients" in output
assert "Train R " in output
assert "Test R " in output
assert "Train RMSE" in output
assert "Test RMSE" in output
```

3 Application

3.1 Calcul du coefficient d'aversion au risque

En utilisant l'équation de Markowitz :

$$U = \mathbb{E}(R_f) - 0.5 \cdot A \cdot \sigma_f^2 \tag{1}$$

1. U l'utilité de cet investissement par rapport à un taux sans risque
2. $\mathbb{E}(R_f)$ l'espérence de rendement du portefeuille
3. A l'aversion au risque qui est le coefficient que l'on cherche à estimer
4. σ_f^2 la volatilité du portefeuille

On pose ainsi le modèle suivant :

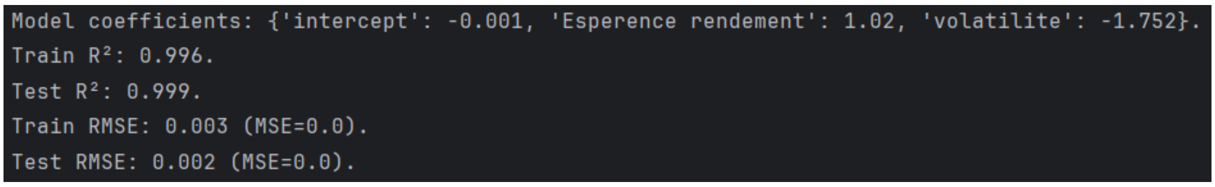
$$U_i = \alpha + \beta_1 \mathbb{E}(R_f) + \beta_2 \sigma_i^2 + \epsilon_i \tag{2}$$

En mettant ce modèle sous forme matricielle, on obtient :

$$U = X\beta + \epsilon \tag{3}$$

$$X = \begin{bmatrix} | & | & | \\ 1 & \mathbb{E}(R_f) & \sigma_f^2 \\ | & | & | \end{bmatrix} \qquad \beta = \begin{bmatrix} \alpha \\ \beta_1 \\ \beta_2 \end{bmatrix}$$

En générent des données pseudos-aléatoire, la régression nous donne :



```
Model coefficients: {'intercept': -0.001, 'Esperence rendement': 1.02, 'volatilite': -1.752}.
Train R²: 0.996.
Test R²: 0.999.
Train RMSE: 0.003 (MSE=0.0).
Test RMSE: 0.002 (MSE=0.0).
```

On remarque ici que $\hat{\beta_2} = -1,752$ et on sait que $\hat{\beta_2} = -\frac{1}{2}\hat{A}$. On en déduit donc que $\hat{A} = 3,504$.

Remarque : On a ici générer $(\epsilon_i)_{1 \leq i \leq n} \underset{i.i.d.}{\sim} \mathcal{N}(0, \sigma^2 = 0.02^2)$

3.2 Prix des logements

Dans cette exemple on s'intéresse au prix d'un logement en fonction de la surface, du nombre de chambre, de la qualité du quartier (qu'on notera de 1 à 5, 5 étant la meilleure qualité de quartier possible) et à l'âge du logement. On aura ainsi l'équation suivante :

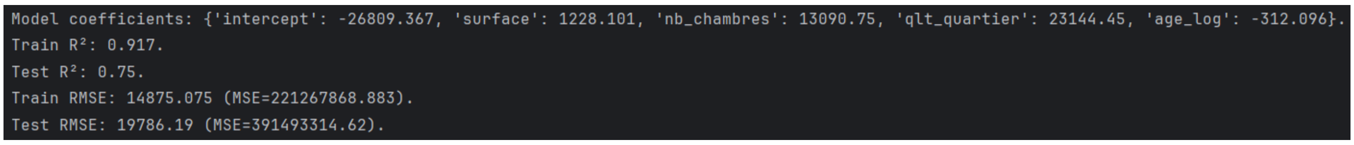
$$prix_i = \alpha + \beta_1 \cdot surface_i + \beta_2 \cdot nb\_chambre_i + \beta_3 \cdot qlt\_quartier_i + \beta_4 \cdot age\_log + \epsilon_i \tag{4}$$

Ce qui revient matriciellement au modèle suivant :

$$Y = X\beta + \epsilon \tag{5}$$

$$X = \begin{bmatrix} | & | & | & | & | \\ 1 & surface & nb\_chambre & qlt\_quartier & age\_log \\ | & | & | & | & | \end{bmatrix} \qquad \beta = \begin{bmatrix} \alpha \\ \beta_1 \\ \beta_2 \\ \beta_3 \\ \beta_4 \end{bmatrix}$$

En estimant ce modèle par MCO, on obtient :



```
Model coefficients: {'intercept': -26809.367, 'surface': 1228.101, 'nb_chambres': 13090.75, 'qlt_quartier': 23144.45, 'age_log': -312.096}.
Train R²: 0.917.
Test R²: 0.75.
Train RMSE: 14875.075 (MSE=221267868.883).
Test RMSE: 19786.19 (MSE=391493314.62).
```

On remarque ici que $\hat{\beta}_1, \hat{\beta}_2, \hat{\beta}_3 > 0$ et $\hat{\beta}_4 < 0$. On a ici générer $(\epsilon_i)_{1 \leq i \leq n} \underset{i.i.d.}{\sim} \mathcal{N}(0, \sigma^2 = 15000^2)$

3.3 Écart de revenus homme femme

Nous nous intéressons ici au revenu en fonction du sexe, de l'âge, de l'expérience, de l'éducation et du secteur . On a ainsi le modèle suivant :

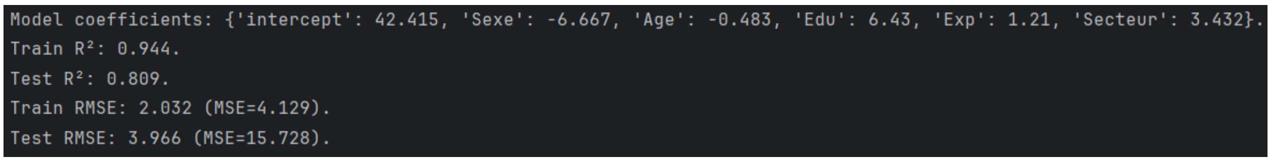
$$salaire_i = \alpha + \beta_1 \cdot sexe_i + \beta_2 \cdot age_i + \beta_3 \cdot edu_i + \beta_4 \cdot exp_i + \beta_5 \cdot secteur_i + \epsilon_i \tag{6}$$

Ce qui revient matriciellement au modèle suivant :

$$Y = X\beta + \epsilon \tag{7}$$

$$X = \begin{bmatrix} | & | & | & | & | & | \\ 1 & sexe & age & edu & exp & secteur \\ | & | & | & | & | & | \end{bmatrix} \qquad \beta = \begin{bmatrix} \alpha \\ \beta_1 \\ \beta_2 \\ \beta_3 \\ \beta_4 \\ \beta_5 \end{bmatrix}$$

En entrant des données pseudo- aléatoire dans notre modèle, on obtient le résultat suivant :



```
Model coefficients: {'intercept': 42.415, 'Sexe': -6.667, 'Age': -0.483, 'Edu': 6.43, 'Exp': 1.21, 'Secteur': 3.432}.
Train R²: 0.944.
Test R²: 0.809.
Train RMSE: 2.032 (MSE=4.129).
Test RMSE: 3.966 (MSE=15.728).
```

3.4 DATA

Dans l'ensemble de ces exemples d'applications, les données sont des données générer de manière à coller plus ou moins avec la réalité. Les variables que l'on cherche à expliquer on été construite comme la somme des varaible expliqué ajouté a un bruit. Afin d'illustrer cette génération de données voici la génération des données de l'écart de revenus homme femmme

```
np.random.seed(42)
n = 50

# ID
ID = np.arange(1, n+1)

# Sexe
Sexe = np.random.choice(["Homme", "Femme"], size=n)

# Age
Age = np.random.randint(22, 61, size=n)

# Education
Education = np.random.choice([0, 1], size=n)

# Experience approximative
Experience = Age - 22 + np.random.randint(-2, 3, size=n)

# Secteur
Secteur = np.random.choice([0, 1], size=n)  # 0=prive, 1=public

# Salaire (pseudo-reel, en  millers euros)
Salaire = (30 +  # base
           5*Education +
           0.8*Experience +
           3*Secteur +
           np.where(Sexe=="Femme",-5,0) +  # effet ecart femme-homme
           np.random.normal(0, 3, n))  # bruit

# DataFrame
df = pd.DataFrame({
    "ID": ID,
    "Sexe": Sexe,
    "Age": Age,
    "Education": Education,
    "Experience": Experience,
    "Secteur": Secteur,
    "Salaire": np.round(Salaire, 2)
})
Sexe = np.where(Sexe == "Femme", 1, 0)  # binaire
X = np.column_stack((Sexe, Age, Education, Experience, Secteur))

# On nomme ici les colonnes correspondants au sexe, age, ...
features_name = ['Sexe', 'Age', 'Edu', 'Exp', 'Secteur']
target_name = ['Salaire']

# Appele a notre fonction 'main'
```

```
main(X, Salaire, features_name, target_name)
```

De manière à coller aux hypothèses des MCO, nottament celle de normalité des résdus, nous avons générer les epsilons de manière à ce qu'ils suivent une loi normal centrée. Dans notre cas :

$$(\epsilon_t)_{1\leq i\leq n} \underset{i.i.d.}{\sim} \mathcal{N}(0, \sigma^2 = 3^2) \tag{8}$$