First ML Project

Build Your First Machine Learning Model with Python

This project takes you from:

Raw Dataset
  ↓
Understand Data
  ↓
Prepare Features
  ↓
Train Model
  ↓
Evaluate Model
  ↓
Make Predictions
  ↓
Save Model

By the end, you will have built a complete beginner-friendly machine learning project using Python, pandas, and scikit-learn.

The project uses the **Iris dataset**, a classic classification dataset available directly through scikit-learn. The same workflow can later be applied to real datasets.

---

1. What You Are Going to Build

You will build a model that predicts the species of an iris flower from measurements of the flower.

The model will receive:

Sepal length
Sepal width
Petal length
Petal width

and predict:

Iris species

The three possible classes are:

Setosa
Versicolor
Virginica

This is a **classification problem** because the model predicts one category from a fixed set of categories.

---

2. Why Start With Iris?

Your first ML project should teach the workflow rather than overwhelm you with a huge dataset.

The Iris dataset is useful because:

- it is small
- it is easy to understand
- the target is already labeled
- the features are numeric
- scikit-learn provides it directly
- you can focus on the ML process

The scikit-learn documentation itself uses Iris to demonstrate the basic process of loading data, splitting it, fitting a model, and

evaluating predictions.

---

3. What You Will Learn

After completing this project, you should understand:

Dataset
Features
Target
Classification
Train/Test Split
Model
Training
Prediction
Accuracy
Confusion Matrix
Pipeline
Model Persistence

You will also learn the most important scikit-learn pattern:

```
model.fit(X_train, y_train)

predictions = model.predict(X_test)
```

---

4. Project Structure

Create a project folder:

```
first_ml_project/
□
□□□ .venv/
□□□ train.py
□□□ predict.py
□□□ model.joblib
□□□ README.md
```

The `.venv` directory is your virtual environment.

`train.py` trains the model.

`predict.py` loads the trained model and makes a prediction.

`model.joblib` stores the trained model.

`README.md` documents the project.

---

5. Create the Virtual Environment

Open your terminal:

```
python -m venv .venv
```

Activate it on Windows:

```
.venv\Scripts\activate
```

On macOS/Linux:

```
source .venv/bin/activate
```

Your terminal should now indicate that the virtual environment is active.

---

6. Install the Libraries

Install the required packages:

```
pip install pandas scikit-learn joblib
```

You can verify the installation:

```
python -c "import pandas, sklearn, joblib; print('Ready')"
```

For current installation guidance, use the official scikit-learn installation documentation.

---

7. Understand the Dataset First

Before writing the model, understand what the data represents.

Each row represents one flower.

Example:

sepal length	sepal width	petal length	petal width	species
5.1	3.5	1.4	0.2	setosa
6.2	2.8	4.8	1.8	virginica

The measurements are the **features**.

The species is the **target**.

Think:

```
Features
   
Model
   
Target prediction
```

---

8. Load the Dataset

Create `train.py`.

Start with:

```
from sklearn.datasets import load_iris

iris = load_iris(as_frame=True)

print(iris.data.head())
```

`load_iris()` gives you the dataset and its metadata.

You can inspect:

```
print(iris.data.head())
print(iris.target.head())
print(iris.target_names)
```

---

9. Understand X and y

Machine learning commonly uses:

X = input features
y = target

For this project:

X = iris.data
y = iris.target

So:

X
→
flower measurements

y
→
flower species

For example:

X:

sepal length
sepal width
petal length
petal width

and:

y:

0
1
2

The numbers represent the three species.

---

10. Inspect the Data

Add:

```
print(X.head())
print(X.shape)
print(X.info())
print(y.value_counts())
```

You should learn to ask:

```
How many rows?
How many features?
What are the data types?
How many classes?
Are values missing?
```

Check missing values:

```
print(X.isna().sum())
```

For this dataset, the features are already clean and numeric, which keeps the first project focused on the ML workflow.

---

11. Look at the Target Names

Run:

```
print(iris.target_names)
```

You will see the species names.

The model works with numeric class labels internally, while you can convert predictions back to human-readable names later.

For example:

```
0 → setosa
1 → versicolor
2 → virginica
```

---

12. Split the Dataset

Do not train and evaluate the model on exactly the same examples.

Split the data:

```
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(
    X,
    y,
    test_size=0.2,
    random_state=42,
    stratify=y
```

)

Now:

80%
□
Training data

20%
□
Testing data

train_test_split() creates random train/test subsets, while random_state makes the split reproducible and stratify can preserve class proportions.

---

13. Why We Need a Test Set

Imagine the model sees:

Flower A □ Setosa
Flower B □ Versicolor
Flower C □ Virginica

during training.

If you test it on those exact same flowers, you do not really know how well it handles new flowers.

Instead:

Training data
□

Model learns

Testing data

Model faces unseen examples

The goal is **generalization**.

---

14. Choose Your First Model

For this project, use:

Logistic Regression

It is a useful first model because the API is simple and it gives you a chance to learn the standard scikit-learn workflow.

Import it:

```
from sklearn.linear_model import LogisticRegression
```

Create it:

```
model = LogisticRegression(max_iter=200)
```

---

15. Train the Model

Now the important line:

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

This tells the model:

> Learn the relationship between these flower measurements and their species.

Conceptually:

```
X_train + y_train
      ↓
    Model
      ↓
Learned parameters
```

---

16. Make Predictions

Now give the model test data:

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

You can inspect them:

```
print(predictions)
```

You can compare them with the actual answers:

```
print(y_test.to_numpy())
```

Think:

```
Actual:
```

```
[0, 1, 2, 1, 0]

Predicted:
[0, 1, 2, 2, 0]
```

Every position represents one test example.

---

17. Evaluate Accuracy

Import:

```
from sklearn.metrics import accuracy_score
```

Calculate:

```
accuracy = accuracy_score(
    y_test,
    predictions
)

print(f"Accuracy: {accuracy:.2%}")
```

If the result is:

```
Accuracy: 96.67%
```

that means the model correctly predicted approximately 96.67% of the test examples.

Do not interpret a single accuracy score as proof that every future prediction will be correct.

---

18. Understand Accuracy

Suppose the test set contains:

`30 flowers`

and the model correctly predicts:

`29 flowers`

Then:

`29 / 30 = 96.67%`

Accuracy answers:

> What percentage of predictions were correct?

For a balanced, simple dataset like Iris, accuracy is an easy starting metric.

For other problems, you may need precision, recall, F1-score, ROC-AUC, or other metrics.

---

19. Confusion Matrix

Accuracy gives you one number.

A confusion matrix shows **where the model's predictions went right or wrong**.

Import:

```
from sklearn.metrics import confusion_matrix

cm = confusion_matrix(
    y_test,
    predictions
)

print(cm)
```

Conceptually:

```
              Predicted
            0   1   2

Actual 0    □   □   □
       1    □   □   □
       2    □   □   □
```

The diagonal represents correct predictions.

The off-diagonal cells represent mistakes.

---

20. Classification Report

You can get several useful metrics at once:

```
from sklearn.metrics import classification_report

print(
```

```
    classification_report(
        y_test,
        predictions,
        target_names=iris.target_names
    )
)
```

This provides metrics such as:

```
Precision
Recall
F1-score
Support
```

This is useful when accuracy alone does not tell the complete story.

---

21. Add a Pipeline

The current dataset does not require complicated preprocessing, but it is useful to learn the production-friendly structure early.

Create:

```
from sklearn.pipeline import make_pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import LogisticRegression
```

Then:

```
model = make_pipeline(
    StandardScaler(),
    LogisticRegression(max_iter=200)
```

)
```

Now:

```
model.fit(X_train, y_train)

predictions = model.predict(X_test)
```

A scikit-learn `Pipeline` combines preprocessing steps and the final estimator into one object and allows them to be fitted and evaluated together, helping prevent test-data leakage.

---

22. Why Use a Pipeline?

Without a pipeline:

```
Scale
  
Train
  
Predict
```

you have to manage each preprocessing step yourself.

With a pipeline:

```
Input
  
StandardScaler
  
LogisticRegression
  
```
```

```
Prediction
```

The complete process becomes one model object.

This becomes much more valuable when a real project has:

```
Missing-value handling
      ↓
Encoding
      ↓
Scaling
      ↓
Feature transformation
      ↓
Model
```

---

23. Your Complete `train.py`

Now combine everything:

```
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.pipeline import make_pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import (
    accuracy_score,
    classification_report,
    confusion_matrix
)
import joblib
```

```
# 1. Load data
iris = load_iris(as_frame=True)

X = iris.data
y = iris.target


# 2. Split data
X_train, X_test, y_train, y_test = train_test_split(
    X,
    y,
    test_size=0.2,
    random_state=42,
    stratify=y
)


# 3. Create model pipeline
model = make_pipeline(
    StandardScaler(),
    LogisticRegression(max_iter=200)
)


# 4. Train
model.fit(X_train, y_train)


# 5. Predict
predictions = model.predict(X_test)


# 6. Evaluate
```

```
accuracy = accuracy_score(
    y_test,
    predictions
)

print(f"Accuracy: {accuracy:.2%}")

print("\nConfusion Matrix:")
print(confusion_matrix(y_test, predictions))

print("\nClassification Report:")
print(
    classification_report(
        y_test,
        predictions,
        target_names=iris.target_names
    )
)



# 7. Save model
joblib.dump(
    model,
    "model.joblib"
)

print("\nModel saved successfully.")
```

Run:

```
python train.py
```

You have now trained your first machine learning model.

---

24. Save the Model

Why save it?

Because you do not want to retrain the model every time someone wants a prediction.

You can save the trained pipeline:

```
joblib.dump(model, "model.joblib")
```

Then load it later:

```
model = joblib.load("model.joblib")
```

scikit-learn documents model persistence as a way to reuse a trained estimator without retraining; its current documentation also discusses security and environment compatibility considerations for serialized models.

---

25. Create `predict.py`

Now create:

```
predict.py
```

Add:

```
import joblib


model = joblib.load("model.joblib")
```

```
prediction = model.predict([
    [5.1, 3.5, 1.4, 0.2]
])

print(prediction)
```

The prediction will be a class number.

---

26. Convert the Prediction to a Species

Instead of printing:

```
[0]
```

load the class names:

```
from sklearn.datasets import load_iris
import joblib


iris = load_iris()

model = joblib.load("model.joblib")

prediction = model.predict([
    [5.1, 3.5, 1.4, 0.2]
])

species = iris.target_names[prediction[0]]

print(f"Predicted species: {species}")
```

The result should identify the flower species.

---

27. Predict Another Flower

Try:

```
prediction = model.predict([
    [6.0, 2.9, 4.5, 1.5]
])

species = iris.target_names[prediction[0]]

print(f"Predicted species: {species}")
```

You have now used your trained model on a new input.

The workflow is:

```
New flower
    
Measurements
    
Saved model
    
Prediction
    
Species
```

---

28. Predict Multiple Flowers

You can provide several flowers at once:

```
flowers = [
    [5.1, 3.5, 1.4, 0.2],
    [6.0, 2.9, 4.5, 1.5],
    [6.5, 3.0, 5.5, 1.8]
]

predictions = model.predict(flowers)

for prediction in predictions:
    print(iris.target_names[prediction])
```

---

29. Add Prediction Probabilities

Classification models can sometimes provide probabilities.

For this model:

```
probabilities = model.predict_proba(flowers)

print(probabilities)
```

You may see something conceptually like:

```
Setosa      0.98
Versicolor   0.02
Virginica    0.00
```

The exact values depend on the trained model and input.

You can select the predicted class:

```
prediction = model.predict(flowers)
```

and the associated probabilities:

```
probabilities = model.predict_proba(flowers)
```

---

30. What Actually Happened?

You wrote relatively little code.

But several ML concepts happened underneath.

```
Dataset
  ↓
Train/Test Split
  ↓
Preprocessing
  ↓
Model Training
  ↓
Learned Parameters
  ↓
Prediction
  ↓
Evaluation
```

This is the core ML pattern you will use repeatedly.

---

31. The `fit()` / `predict()` Mental Model

Remember these two methods:

`fit()`

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

Means:

> Learn from examples.

`predict()`

```
model.predict(X_test)
```

Means:

> Use what you learned to make predictions.

So:

```
fit()
 
learn

predict()
 
use what was learned
```

---

32. What Is the Model Learning?

The model is not memorizing the sentence:

`"This is flower number 17."`

It is learning relationships between measurements and classes.

For example:

```
Petal length
Petal width
    
Species
```

The model finds patterns that help distinguish:

```
Setosa
Versicolor
Virginica
```

This is the basic idea behind supervised learning.

---

33. Try a Different Model

Now experiment.

Replace Logistic Regression with:

```
from sklearn.tree import DecisionTreeClassifier

model = DecisionTreeClassifier(
    random_state=42
)
```

Train:

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

Predict:

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

Evaluate:

```
print(
    accuracy_score(
        y_test,
        predictions
    )
)
```

Now you have performed your first **model comparison**.

---

34. Try Random Forest

Another experiment:

```
from sklearn.ensemble import RandomForestClassifier
```

```
model = RandomForestClassifier(
    n_estimators=100,
    random_state=42
)
```

Then:

```
model.fit(X_train, y_train)

predictions = model.predict(X_test)

print(
    accuracy_score(
        y_test,
        predictions
    )
)
```

You can compare:

```
Logistic Regression
        
Decision Tree
        
Random Forest
```

The goal is not simply to chase the highest number.

You are learning how different models behave.

---

35. Create a Simple Model Comparison

```
from sklearn.linear_model import LogisticRegression
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import accuracy_score


models = {
    "Logistic Regression":
        LogisticRegression(max_iter=200),

    "Decision Tree":
        DecisionTreeClassifier(random_state=42),

    "Random Forest":
        RandomForestClassifier(
            n_estimators=100,
            random_state=42
        )
}


for name, model in models.items():

    model.fit(X_train, y_train)

    predictions = model.predict(X_test)

    score = accuracy_score(
        y_test,
        predictions
    )

    print(
        f"{name}: {score:.2%}"
```

```
)
```

This introduces an important ML habit:

> Compare models using the same evaluation setup.

---

36. Add Cross-Validation

A single train/test split can give you only one view of performance.

Try cross-validation:

```
from sklearn.model_selection import cross_val_score

scores = cross_val_score(
    model,
    X,
    y,
    cv=5
)

print(scores)
print(scores.mean())
```

Conceptually:

```
Fold 1 → score
Fold 2 → score
Fold 3 → score
Fold 4 → score
```

```
Fold 5 □ score

    □

Average score
```

Cross-validation is useful when comparing models and estimating how consistently they perform across different splits.

---

37. Project Workflow

Your finished project now looks like:

```
          □□□□□□□□□□□□□□□□□□
          □ Iris Dataset □
          □□□□□□□□□□□□□□□□□□
                ↓
          Inspect Dataset
                ↓
          Define X and y
                ↓
          Train/Test Split
                ↓
          Build Pipeline
                ↓
           Train Model
                ↓
           Predictions
                ↓
           Evaluation
                ↓
          Model Comparison
                ↓
```

Save Model
↓
Load Model Later
↓
New Predictions

---

38. What Makes This a Real ML Project?

A project becomes more meaningful when you can explain every stage.

You should be able to answer:

What problem am I solving?

What is my target?

What are my features?

Why did I split the data?

Why did I choose this model?

How did I evaluate it?

What does the accuracy mean?

What mistakes does the model make?

How would I improve it?

How would I deploy it?

If you cannot explain these, copying the code is not enough.

---

39. Common Beginner Mistakes

Mistake 1: Training on the entire dataset

Avoid:

```
model.fit(X, y)

predictions = model.predict(X)
```

This evaluates the model on data it already saw.

Instead:

```
model.fit(X_train, y_train)

predictions = model.predict(X_test)
```

Mistake 2: Looking only at training accuracy

A model can perform extremely well on training data and poorly on unseen data.

Always evaluate on held-out data.

Mistake 3: Changing the test set repeatedly

Do not repeatedly modify your model based on the final test result.

Use validation or cross-validation while experimenting.

Keep the test set for final evaluation.

Mistake 4: Ignoring preprocessing

As datasets become more complicated, preprocessing becomes important.

Use pipelines when preprocessing is part of the model workflow.

Mistake 5: Treating accuracy as universal

Accuracy works as a simple starting metric here.

For problems such as fraud detection, medical screening, or spam detection, other metrics may matter more.

Mistake 6: Saving only the model and forgetting preprocessing

If your production model requires preprocessing, save the **complete pipeline**, not just the final estimator.

For example:

```
joblib.dump(
    pipeline,
    "model.joblib"
```

)
```

Then loading the file restores the complete prediction workflow.

---

## 40. Improve the Project

Your first version is intentionally simple.

Now improve it.

### Level 1

Add:

```
Dataset inspection
Model comparison
Confusion matrix
Classification report
```

Level 2

Add:

```
Cross-validation
Hyperparameter tuning
```

Level 3

Create:

```
FastAPI endpoint
```
```

Level 4

Create a small frontend:

```
User enters measurements
      ↓
Frontend
      ↓
FastAPI
      ↓
Saved model
      ↓
Prediction
```

Level 5

Deploy it.

Now your ML project becomes an actual application.

---

41. Turn It Into a Web App

Imagine the interface:

```
Iris Species Predictor

Sepal Length:  [ 5.1 ]

Sepal Width:   [ 3.5 ]

Petal Length:  [ 1.4 ]
```

```
Petal Width:   [ 0.2 ]

      [ Predict ]

Prediction:
Setosa
```

The architecture:

```
Browser
  □
FastAPI
  □
model.joblib
  □
Prediction
  □
JSON response
  □
Browser
```

This is where machine learning starts connecting with normal software engineering.

---

42. Suggested Project Structure After Expansion

```
first_ml_project/
□
□□□ app/
□   □□□ __init__.py
□   □□□ main.py
□   □□□ model.py
```

```
□
□□□ data/
□   □□□ README.md
□
□□□ models/
□   □□□ iris_model.joblib
□
□□□ notebooks/
□   □□□ exploration.ipynb
□
□□□ tests/
□   □□□ test_model.py
□
□□□ train.py
□□□ predict.py
□□□ requirements.txt
□□□ README.md
```

You do not need this structure for the first version.

It becomes useful as the project grows.

---

43. What You Should Put in README.md

A good project README should explain:

```
Project name

Problem

Dataset
```

Features

Target

Model

Training process

Evaluation

Example prediction

How to install

How to run

Future improvements

Example:

```
# Iris Species Predictor

A beginner machine learning project that
predicts iris flower species from four
flower measurements.

## Model

Logistic Regression with StandardScaler.

## Features

- Sepal length
- Sepal width
```

- Petal length
- Petal width

Target

Iris species.

Run

python train.py

python predict.py

---

44. Mini Challenge

Build the same project without looking at the complete code.

Your checklist:

[ ] Create virtual environment

[ ] Install dependencies

[ ] Load Iris dataset

[ ] Inspect X and y

[ ] Split train/test data

[ ] Create model

[ ] Train model

[ ] Make predictions

[ ] Calculate accuracy

[ ] Generate confusion matrix

[ ] Generate classification report

[ ] Save model

[ ] Load model

[ ] Predict a new flower

If you can complete this without copying every line, you have understood the workflow.

---

45. Exercises

Exercise 1

Replace Logistic Regression with:

`DecisionTreeClassifier`

Compare the result.

Exercise 2

Try:

`RandomForestClassifier`

Compare it with your previous models.

Exercise 3

Use 30% test data instead of 20%.

`test_size=0.3`

Observe how the result changes.

Exercise 4

Remove the `StandardScaler`.

Does Logistic Regression still work?

Compare the result.

Exercise 5

Use 5-fold cross-validation.

Compare the average score with your single test-set score.

Exercise 6

Create a function:

```
def predict_species(
    sepal_length,
    sepal_width,
    petal_length,
    petal_width
):
    ...
```

It should return:

```
setosa
versicolor
virginica
```

---

46. Five Minute Challenge

Without opening your previous code, write these five lines conceptually:

```
1. Load data
2. Split data
3. Create model
4. Fit model
5. Predict
```

Then write the Python equivalent.

The goal is to remember the workflow rather than memorize a complete project.

---

47. Interview Questions

Beginner

1. What is the Iris dataset?
2. Is Iris classification or regression?
3. What are the features in this project?
4. What is the target?
5. Why do we split data?
6. What does `fit()` do?
7. What does `predict()` do?
8. What is accuracy?

Intermediate

9. Why should we keep test data separate?
10. What is a confusion matrix?
11. Why use a pipeline?
12. What is cross-validation?
13. Why save a trained model?
14. Why should preprocessing be part of the saved prediction workflow?
15. What is overfitting?

Project-Based

16. Why did you choose Logistic Regression?
17. Which other models did you test?
18. How did you evaluate your model?
19. What would you do if accuracy were poor?
20. How would you deploy this model?
21. How would you turn this project into an API?
22. How would you monitor it after deployment?

---

48. Cheat Sheet

LOAD
↓
load_iris()

FEATURES
↓
X = iris.data

TARGET
↓
y = iris.target

SPLIT
↓
train_test_split()

MODEL
↓
LogisticRegression()

PIPELINE
↓
StandardScaler()
+
LogisticRegression()

TRAIN
↓
model.fit(X_train, y_train)

PREDICT

→
model.predict(X_test)

EVALUATE
→
accuracy_score()

DEEPER EVALUATION
→
confusion_matrix()
classification_report()

SAVE
→
joblib.dump()

LOAD
→
joblib.load()

NEW PREDICTION
→
model.predict(new_data)

---

49. The Core Mental Model

Remember:

MACHINE LEARNING PROJECT

PROBLEM
→

```
DATA
 
FEATURES + TARGET
 
TRAIN / TEST
 
MODEL
 
FIT
 
PREDICT
 
EVALUATE
 
SAVE
 
NEW PREDICTIONS
```

And the core scikit-learn pattern:

```
model.fit(X_train, y_train)

predictions = model.predict(X_test)
```

Everything else builds around these ideas.

---

50. What You Have Built

You now have a complete first ML project that can:

 Load a dataset
 Understand features and target

 Split data
 Preprocess data
 Train a classification model
 Make predictions
 Evaluate performance
 Inspect classification errors
 Compare models
 Save the trained model
 Load the model later
 Predict new examples

More importantly, you have practiced the complete cycle:

Data
 Train
 Predict
 Evaluate
 Improve
 Save
 Use

That is the foundation for larger machine learning projects.

---

51. Next Step

After this project, move from a clean educational dataset toward a **real-world dataset**.

A useful progression is:

Iris

Real CSV Dataset

Data Cleaning

Feature Engineering

Model Comparison

Cross-Validation

Hyperparameter Tuning

Model Persistence

FastAPI

Deployment

The objective is not to build the most complicated model.

The objective is to become comfortable taking a problem from **data model prediction usable application**.

Visit haas.dev for more resources and guides.

Website:
https://dev-roast-app.vercel.app

Resources:
https://dev-roast-app.vercel.app/resources

Official references:
https://scikit-learn.org/stable/

https://scikit-learn.org/stable/getting_started.html