Practical Work: Regression ML Pipeline

Louis Fippo Fitime, Claude Tinku, Kerolle Sonfack
Department of Computer Engineering, ENSPY

October 11, 2025

TP Objective

- Master the theoretical trade-offs (**Bias vs Variance**) inherent in modeling.
- Implement and compare regularized linear regression models (**Ridge** and **LASSO**).
- Estimate optimal parameters () using **Cross-Validation**.
- Understand and analyze the concept of **Sparsity** introduced by LASSO.
- Initiate the management of the model life cycle (**MLOps**) through **Packaging** and **Tracking (MLflow)**.

---

1 Phase I: Advanced Linear Regression and Trade-off Analysis

1.1 Data Preparation and OLS Model

We will use the Diabetes dataset, which aims to predict a quantitative measure of disease progression from ten physiological variables (age, sex, BMI, etc.).

- **Task 1.1:** Load the dataset. Split the data into training and test sets (80
- **Task 1.2: OLS Regression:** Train an Ordinary Least Squares (OLS) Linear Regression model on the normalized data. Evaluate its performance in terms of R 2 and Root Mean Squared Error (RMSE).

```
Code Python 1.1 & 1.2: OLS
import numpy as np
from sklearn.datasets import load_diabetes
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import LinearRegression
```

```
from sklearn.metrics import mean_squared_error, r2_score

Data loading and preparation
X, y = load_diabetes(return_X_y=True)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=4

Normalization (already pre-processed, but we apply a standard for consistency)
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)

OLS Regression
ols_reg = LinearRegression()
ols_reg.fit(X_train_scaled, y_train)
y_pred_ols = ols_reg.predict(X_test_scaled)

OLS Evaluation
rmse_ols = np.sqrt(mean_squared_error(y_test, y_pred_ols))
r2_ols = r2_score(y_test, y_pred_ols)

print(f"OLS: RMSE={rmse_ols:.2f}, R2={r2_ols:.2f}")
print("Coefficients OLS:", ols_reg.coef_)
```

1.2 Bias, Variance, and Regularization

- **Task 1.3: Theory:** Describe the relationship between **Bias** and **Variance** in Machine Learning. Explain how adding a regularization term (penalty) modifies this trade-off and prevents **overfitting**.
- **Task 1.4: Ridge Regression ($_2$):** Implement a Ridge model. Justify why the $_2$ penalty tends to reduce the magnitude of coefficients, but does not completely zero them out.

```
Code Python 1.4: Ridge Regression
from sklearn.linear_model import Ridge
from sklearn.model_selection import cross_val_score

Training with an initial alpha value
alpha_initial = 1.0
ridge_reg = Ridge(alpha=alpha_initial)
ridge_reg.fit(X_train_scaled, y_train)

Evaluation by Cross-Validation (5-folds)
scores_ridge = cross_val_score(ridge_reg, X_train_scaled, y_train, scoring='neg_root_me
rmse_cv_ridge = -scores_ridge.mean()

print(f"\nRidge (alpha={alpha_initial}): RMSE CV={rmse_cv_ridge:.2f}")
print("Coefficients Ridge:", ridge_reg.coef_)
```

1.3 Sparsity and Hyperparameter Tuning

- **Task 1.5: LASSO Regression ($_1$):** Implement a LASSO model. Explain the concept of **Sparsity**. Why can the $_1$ penalty (unlike $_2$) zero out certain coefficients, thereby implicitly performing **feature selection**?
- **Task 1.6: Tuning with Cross-Validation:** Use `LassoCV` and `RidgeCV` (which include internal cross-validation) to find the optimal values of the hyperparameter for each model.

```
Code Python 1.6: Tuning with CV for Ridge and LASSO
from sklearn.linear_model import RidgeCV, LassoCV

Ridge Tuning
ridge_cv = RidgeCV(alphas=np.logspace(-3, 2, 100), cv=10) # Tests 100 alpha values
ridge_cv.fit(X_train_scaled, y_train)
alpha_ridge = ridge_cv.alpha_

LASSO Tuning
lasso_cv = LassoCV(alphas=np.logspace(-3, 0, 100), cv=10, max_iter=10000)
lasso_cv.fit(X_train_scaled, y_train)
alpha_lasso = lasso_cv.alpha_

print(f"\nOptimal Alpha for Ridge: {alpha_ridge:.4f}")
print(f"Optimal Alpha for LASSO: {alpha_lasso:.4f}")

Final LASSO Model
lasso_final = LassoCV(alphas=[alpha_lasso], cv=10, max_iter=10000)
lasso_final.fit(X_train_scaled, y_train)

print("\nFinal LASSO Coefficients (Sparsity analysis):")
print(np.round(lasso_final.coef_, 4))
print(f"Number of zero coefficients: {np.sum(lasso_final.coef_ == 0)}")
```

---

2 Phase II: MLOps: From Modeling to Production

The crucial step after modeling is deployment into production and managing the model's life cycle.

2.1 Model Packaging and Dependencies

- **Task 2.1: Packaging:** Save the best-trained model (the one that offers the best performance on the test set or the best Sparsity/Performance balance), as well as the trained `StandardScaler` object, using `joblib`. Explain why it is imperative to save the **scaler**.
- **Task 2.2: Dependencies:** Identify the dependencies required for running this model in production (`scikit-learn`, `numpy`, `pandas`, `joblib`).

```
Code Python 2.1: Packaging (with the best model, here LASSO)
import joblib

Saving the transformation object and the final model
joblib.dump(scaler, 'scaler.pkl')
joblib.dump(lasso_final, 'lasso_model.pkl')

print("\nScaler and LASSO model saved for deployment.")

Example of a prediction function in production
def predict_new_data(raw_data, model_path='lasso_model.pkl', scaler_path='scaler.pkl')
"""Simulates the prediction pipeline in production."""
try:
loaded_scaler = joblib.load(scaler_path)
loaded_model = joblib.load(model_path)
except FileNotFoundError:
print("Error: Model or scaler files not found.")
return None

Step 1: Raw data transformation
scaled_data = loaded_scaler.transform([raw_data])

Step 2: Prediction
prediction = loaded_model.predict(scaled_data)[0]
return prediction

Testing the prediction function with a point from the test set
example_raw_input = X_test[0]
predicted_value = predict_new_data(example_raw_input)
print(f"Prediction for the test example: {predicted_value:.2f}")
```

2.2 MLflow for Lifecycle Management (Simulated)

- **Task 2.3: Tracking: MLflow** is a key platform for managing the entire ML life cycle. Write the necessary code to start an "Experiment," logging the hyperparameters () and the final metric (RMSE) for the three models (OLS, Ridge, LASSO).

```
Code Python 2.3: Tracking with MLflow (API Simulation)

Note: MLflow must be installed (pip install mlflow) and the server must be started.
import mlflow

Configuration of the experiment
mlflow.set_experiment("TP_Advanced_Regression")

def log_model_results(model_name, params, metrics, model):
"""Logs the results of a model in MLflow."""
with mlflow.start_run(run_name=model_name):
```

```
Hyperparameter Logging
mlflow.log_params(params)

# Metric Logging
mlflow.log_metrics(metrics)

# Model Logging (artifact)
# mlflow.sklearn.log_model(model, "model")

print(f"MLflow Run '{model_name}' logged.")

Example logging for the LASSO model
lasso_params = {"alpha": alpha_lasso, "solver": "cd", "penalty": "l1"}
y_pred_lasso = lasso_final.predict(X_test_scaled)
rmse_lasso = np.sqrt(mean_squared_error(y_test, y_pred_lasso))
lasso_metrics = {"test_rmse": rmse_lasso, "test_r2": r2_score(y_test, y_pred_lasso)}

log_model_results("LASSO_Optimal", lasso_params, lasso_metrics, lasso_final)
```

2.3 Containerization (Docker)

- **Task 2.4: Dockerfile (Theoretical):** Provide a basic `Dockerfile` that allows containerizing a small web API (for example, based on Flask/FastAPI) exposing the `lasso_model.pkl` model for predictions. Explain the role of the commands `FROM`, `COPY`, and `CMD`.

```
Dockerfile: Containerization Structure

This is a theoretical Dockerfile example.
1. Light Python base image
FROM python:3.9-slim

2. Define the working directory
WORKDIR /app

3. Copy dependency file and install packages
(Assume requirements.txt contains sklearn, numpy, joblib, flask)
COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt

4. Copy the saved model (lasso_model.pkl, scaler.pkl) and the API code (app.py)
COPY . /app

5. Command to start the API application when launching the container
(where app.py is a Flask/FastAPI script that exposes the prediction function)
CMD ["python", "app.py"]
```

3 Conclusion and Post-Production Perspectives

- **Task 3.1: MLOps Synthesis:** Write a brief note explaining why the use of **Git**, **MLflow**, and **Docker** is fundamental for setting up robust **CI/CD** chains (Continuous Integration and Continuous Deployment) in ML engineering.
- **Task 3.2: Monitoring and Debugging:** Cite two crucial metrics or aspects to monitor once the model is in production (**Post-Productionizing**), besides the RMSE. (Hint: Think about data drift).