Sri Sivasubramaniya Nadar College of Engineering, Chennai

(An Autonomous Institution Affiliated to Anna University)

Department of Computer Science and Engineering

Theory Assignment 1

Name: Nithish Ra
Register No: 3122237001033

Degree & Branch: M. Tech (Integrated) Computer Science & Engineering
Semester: V
Subject Code & Name: ICS1502 – Introduction to Machine Learning
Academic Year: 2025–2026 (Odd) Batch: 2023–2028

1. Regression: Mobile Phone Price Prediction

In this task, we implement and evaluate **Linear Regression** models for predicting mobile phone prices. We use the dataset `Mobile-Price-Prediction-cleaned_data.csv`, where the target variable is "Price".

1.1 Data Representation

The dataset was split into training and testing sets using an 80:20 split. Let:

$$X \in R^{m \times d}, \quad y \in R^{m \times 1}$$

where m is the number of samples and d is the number of features. A bias term was added to X.

```python
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import mean_squared_error, r2_score

data = pd.read_csv("/Mobile-Price-Prediction-cleaned_data.csv")
print("Dataset shape:", data.shape)

# Separate features (X) and target (y)
X = data.drop("Price", axis=1).values    # shape (m, d)
y = data["Price"].values.reshape(-1, 1)   # shape (m, 1)

m, d = X.shape
print(f"m={m}, d={d}")

# Split the data
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size
    =0.2, random_state=42)
m_train = X_train.shape[0]
m_test  = X_test.shape[0]
print("Train shape:", X_train.shape, " Test shape:", X_test.shape)

# Function to add bias (x0=1) column
def add_bias(X):
    return np.concatenate([np.ones((X.shape[0], 1)), X], axis=1)
```

1.2 Closed-form Solution

Using the Moore–Penrose pseudoinverse, we calculate θ directly using the Normal Equation, implemented via `np.linalg.pinv`.

$$\theta = (X^T X)^{-1} X^T y$$

This gives the optimal parameters directly on the unscaled data.

```python
# Add bias to the unscaled training and test data
X_train_bias = add_bias(X_train)
X_test_bias  = add_bias(X_test)
```

```python

# Calculate theta using pseudoinverse
theta_closed = np.linalg.pinv(X_train_bias) @ y_train   # shape (d+1, 1)
y_pred_closed = X_test_bias @ theta_closed

# Evaluate performance
mse_closed = mean_squared_error(y_test, y_pred_closed)
r2_closed = r2_score(y_test, y_pred_closed)
print("\nClosed-form (pinv) -- unscaled: MSE = {:.4f}, R2 = {:.4f}".format(mse_closed, r2_closed))
```

1.3 Gradient Descent

We implemented batch gradient descent with L2 regularization support (controlled by λ). The non-regularized update rule is:

$$\theta := \theta - \alpha \left(\frac{1}{m} X^T (X\theta - y) \right)$$

where α is the learning rate. Convergence was monitored using loss plots. For this step, we standardized the data using `StandardScaler` before applying GD, as it converges much more reliably and quickly.

```python
# Standardize features for GD
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled  = scaler.transform(X_test)

# Add bias to standardized data
X_train_scaled_bias = add_bias(X_train_scaled)
X_test_scaled_bias  = add_bias(X_test_scaled)

# Note: The provided code re-uses the variable names X_train_bias/X_test_bias.
# For clarity, we will rename them here to reflect they are scaled.
X_train_bias_scaled_gd = X_train_scaled_bias
X_test_bias_scaled_gd  = X_test_scaled_bias

def gradient_descent(X, y, lr=0.01, epochs=2000, lam=0.0, tol=1e-8, verbose=False):
    m, n = X.shape   # m samples, n = d+1 params
    theta = np.zeros((n,1))
    prev_loss = np.inf
    losses = []
    for it in range(epochs):
        # predictions
        preds = X @ theta                # (m,1)
        error = preds - y                # (m,1)
        # gradient: (1/m) X^T (X theta - y) + (lambda/m)*[0; theta_1:]
        grad = (1.0/m) * (X.T @ error) + (lam/m) * np.r_[[[0]], theta[1:]]  # do not reg bias
        theta = theta - lr * grad
        loss = (1.0/(2*m)) * np.sum(error**2) + (lam/(2*m))*np.sum(theta[1:]**2)
        losses.append(loss)
        if it % 200 == 0 and verbose:
            print(f"GD iter {it}, loss {loss:.6f}")
        if abs(prev_loss - loss) < tol:
            if verbose:
                print(f"Converged at iter {it}, loss diff {abs(
                    prev_loss-loss):.2e}")
            break
        prev_loss = loss
    return theta, losses

# Run GD without regularization (lambda=0) on STANDARDIZED data
theta_gd, losses_gd = gradient_descent(X_train_bias_scaled_gd, y_train,
     lr=0.01, epochs=10000, lam=0.0, tol=1e-10, verbose=False)
y_pred_gd = X_test_bias_scaled_gd @ theta_gd
mse_gd = mean_squared_error(y_test, y_pred_gd)
r2_gd = r2_score(y_test, y_pred_gd)
print("Gradient Descent -- standardized: MSE = {:.4f}, R2 = {:.4f}".
    format(mse_gd, r2_gd))
```

1.4 Ridge Regression (L2 Regularization)

The closed-form ridge solution modifies the Normal Equation to prevent matrix singularity and control overfitting:

$$\theta = (X^T X + \lambda I)^{-1} X^T y$$

We implement this using `np.linalg.solve` for better numerical stability than direct inversion. We tested this function and the GD function (with $\lambda > 0$) on both standardized and unstandardized data.

```
def ridge_closed_form(X, y, lam=1.0):
    n = X.shape[1]
    D = np.eye(n)
    D[0,0] = 0.0 # Do not regularize the bias term
    A = X.T @ X + lam * D
    theta = np.linalg.solve(A, X.T @ y)   # more stable than inv
    return theta

# Test 1: Ridge Closed-Form on UNSCALED data
theta_ridge_unstd = ridge_closed_form(X_train_bias, y_train, lam=1.0)
y_pred_ridge_unstd = X_test_bias @ theta_ridge_unstd
print("\nRidge (lam=1) unstandardized: MSE = {:.4f}, R2 = {:.4f}".
    format(
    mean_squared_error(y_test, y_pred_ridge_unstd), r2_score(y_test,
        y_pred_ridge_unstd)))

# Test 2: Ridge using Gradient Descent on STANDARDIZED data (
    recommended)
theta_ridge_gd_std, _ = gradient_descent(X_train_bias_scaled_gd,
    y_train, lr=0.01, epochs=5000, lam=1.0, verbose=False)
y_pred_ridge_gd_std = X_test_bias_scaled_gd @ theta_ridge_gd_std
print("Ridge GD (lam=1) standardized: MSE = {:.4f}, R2 = {:.4f}".format
    (
    mean_squared_error(y_test, y_pred_ridge_gd_std), r2_score(y_test,
        y_pred_ridge_gd_std)))
```

1.5 Performance Plots

Plots were generated to verify model performance and convergence.

- Predicted vs. Actual plots showed good alignment (using unscaled predictions for closed-form and GD-scaled predictions).
- Loss vs. Iterations plot confirmed gradient descent convergence.
- λ vs. (MSE, R^2) plots demonstrated the effect of regularization strength on standardized data.

```python
# 1. GD Training Loss Plot
plt.figure(figsize=(5,3))
plt.plot(losses_gd)
plt.title("GD Training Loss (standardized)")
plt.xlabel("Iteration")
plt.ylabel("Loss")
plt.tight_layout()
plt.show()

# 2. Predicted vs Actual Plot
plt.figure(figsize=(6,6))
plt.scatter(y_test, y_pred_closed, alpha=0.6, label="Closed-form (pinv)
    - unscaled")
plt.scatter(y_test, y_pred_gd, alpha=0.6, label="Gradient Descent -
    scaled")
minv = min(y_test.min(), y_pred_closed.min(), y_pred_gd.min())
maxv = max(y_test.max(), y_pred_closed.max(), y_pred_gd.max())
plt.plot([minv, maxv], [minv, maxv], 'r--')
plt.xlabel("Actual Price")
plt.ylabel("Predicted Price")
plt.legend()
plt.title("Predicted vs Actual")
plt.show()

# 3. Effect of Lambda (on standardized features)
def evaluate_ridge_on_scaled(lam):
    theta = ridge_closed_form(X_train_scaled_bias, y_train, lam=lam)
    preds = X_test_scaled_bias @ theta
    return theta, preds

lams = [0.0, 0.01, 0.1, 1, 10, 100]
mse_vals = []
r2_vals  = []
thetas = {} # Store thetas for feature importance
for lam in lams:
    theta, preds = evaluate_ridge_on_scaled(lam)
    mse_vals.append(mean_squared_error(y_test, preds))
    r2_vals.append(r2_score(y_test, preds))
    thetas[lam] = theta

print("\nRidge on STANDARDIZED features (summary):")
for lam,mse,r2 in zip(lams, mse_vals, r2_vals):
    print(f"  lambda={lam:>6} : MSE={mse:.4f}, R2={r2:.4f}")

# 4. Lambda vs Metrics Plot
plt.figure(figsize=(6,4))
plt.plot(lams, mse_vals, marker=’o’, label="MSE")
# plt.plot(lams, r2_vals, marker=’s’, label="R2") # R2 is on different scale, plotting MSE only
plt.xscale(’log’)
plt.xlabel("Lambda (  )")
plt.ylabel("MSE")
plt.title("Effect of    on Ridge MSE (standardized)")
plt.legend()
plt.show()
```

1.6 Feature Importance

From standardized ridge coefficients (using $\lambda = 1.0$), we ranked features by their absolute coefficient values. This highlights which mobile phone attributes most influence the price prediction. Because the features are standardized (same scale), the magnitude of their coefficients is a direct proxy for importance.

```
# ***FIX: We must define feature_names from the dataframe columns***
feature_names = data.drop("Price", axis=1).columns

lam_for_importance = 1.0
theta_std = thetas[lam_for_importance].flatten()   # shape (d+1,)
coef_std = theta_std[1:]   # skip bias term

# Display numeric sorted importances
importance_df = pd.DataFrame({
    "feature": feature_names,
    "coef": coef_std,
    "abs_coef": np.abs(coef_std)
}).sort_values(by="abs_coef", ascending=False)

print("\nFeature importance (standardized coefficients) sorted:")
print(importance_df.to_string(index=False))

# Plot feature importances
plt.figure(figsize=(10,4))
plt.bar(importance_df[’feature’], importance_df[’coef’])
plt.xticks(rotation=90)
plt.title(f"Feature importance (coefficients)     standardized features (  ={lam_for_importance})")
plt.tight_layout()
plt.show()
```

Summary

Closed-form and gradient descent gave consistent results, especially when GD was run on standardized data. Ridge regression provided better generalization. Standardization is crucial for both GD convergence and for interpreting feature importances from model coefficients.

2. Linear Classification: Bank Note Authentication

We evaluated the suitability of logistic regression on the `BankNote_Authentication.csv` dataset.

2.1 Dataset Split

The dataset was divided into 70% training and 30% testing, using stratification to maintain the class balance in both splits.

```python
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score
from mpl_toolkits.mplot3d import Axes3D

data_clf = pd.read_csv("/content/BankNote_Authentication.csv")
print("Dataset shape:", data_clf.shape)

X = data_clf.drop("class", axis=1)
y = data_clf["class"]

# Split into 70% train, 30% test
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.3, random_state=42, stratify=y
)

print("Training set size:", X_train.shape)
print("Test set size:", X_test.shape)
```

2.2 Logistic Regression without and with Regularization

We compared a standard L2 regularized model (C=1.0) against a model simulating no regularization (by setting $C = 1e10$, which makes the regularization penalty $\lambda = 1/C$ near zero).

- Without regularization: high training accuracy, slight overfitting observed.
- With L2 regularization: improved generalization, better balance between training and test accuracy.

```python
# 1. Approx "No Regulation" by using a massive C value
clf_no_reg = LogisticRegression(penalty="l2", C=1e10, solver="lbfgs",
    max_iter=1000)
clf_no_reg.fit(X_train, y_train)

y_pred_train_no_reg = clf_no_reg.predict(X_train)
y_pred_test_no_reg = clf_no_reg.predict(X_test)

print("Training Accuracy (No Reg approx):", accuracy_score(y_train,
    y_pred_train_no_reg))
print("Test Accuracy (No Reg approx):", accuracy_score(y_test,
    y_pred_test_no_reg))


# 2. Standard L2 Regularization (C=1.0)
```

```python
clf_l2 = LogisticRegression(penalty="l2", C=1.0, solver="lbfgs", max_iter=1000)
clf_l2.fit(X_train, y_train)

y_pred_train_l2 = clf_l2.predict(X_train)
y_pred_test_l2 = clf_l2.predict(X_test)

print("\nTraining Accuracy (L2, C=1.0):", accuracy_score(y_train, y_pred_train_l2))
print("Test Accuracy (L2, C=1.0):", accuracy_score(y_test, y_pred_test_l2))
```

2.3 Accuracy vs. Regularization Strength

By varying $\lambda = 1/C$, we plotted train and test accuracies. We tested λ values from 10^{-4} to 10^4.

- Small λ (weak regularization): risk of overfitting (Train acc ¿ Test acc).
- Large λ (strong regularization): underfitting (Both accuracies drop as the model becomes too simple).

Optimal performance was observed at intermediate λ values.

```python
lambdas = np.logspace(-4, 4, 20)  #    = 1/C
train_acc = []
test_acc = []

for lam in lambdas:
    C_val = 1.0 / lam
    model = LogisticRegression(penalty="l2", C=C_val, solver="lbfgs", max_iter=1000)
    model.fit(X_train, y_train)
    train_acc.append(accuracy_score(y_train, model.predict(X_train)))
    test_acc.append(accuracy_score(y_test, model.predict(X_test)))

plt.figure(figsize=(8,5))
plt.semilogx(lambdas, train_acc, label="Train Accuracy", marker="o")
plt.semilogx(lambdas, test_acc, label="Test Accuracy", marker="s")
plt.xlabel("   (Regularization strength = 1/C)")
plt.ylabel("Accuracy")
plt.title("Train & Test Accuracy vs   ")
plt.legend()
plt.grid(True)
plt.show()
```

2.4 Data Visualization

A 3D scatter plot using `variance`, `skewness`, and `curtosis` illustrated that the classes are highly (though perhaps not perfectly linearly) separable.

```python
fig = plt.figure(figsize=(8,6))
ax = fig.add_subplot(111, projection='3d')

# Choose three features
```

```python
ax.scatter(X_train["variance"], X_train["skewness"], X_train["curtosis"
    ],
           c=y_train, cmap="bwr", alpha=0.7)

ax.set_xlabel("Variance")
ax.set_ylabel("Skewness")
ax.set_zlabel("Curtosis")
ax.set_title("3D Visualization of Bank Note Data")
plt.show()
```

2.5 Outlier Analysis

Artificial outliers were introduced by adding significant random noise to 20 data points in the training set. Impact: Training accuracy remained high (as the model tried to fit the noise), but test accuracy dropped, showing that the decision boundary was distorted by the outliers. This demonstrates that logistic regression is sensitive to outliers, which reduces its generalization capability.

```python
# Copy dataset
X_outlier = X_train.copy()
y_outlier = y_train.copy()

# Introduce outliers by shifting some points
n_outliers = 20
rng = np.random.RandomState(42)
outlier_indices = rng.choice(len(X_outlier), size=n_outliers, replace=
    False)

# Add large noise to the selected outlier indices
X_outlier.iloc[outlier_indices] = X_outlier.iloc[outlier_indices] + rng
    .normal(20, 5, X_outlier.shape[1])

print("Introduced", n_outliers, "outliers.")

# Fit the standard L2 model on the noisy data
clf_outlier = LogisticRegression(penalty="l2", C=1.0, solver="lbfgs",
    max_iter=1000)
clf_outlier.fit(X_outlier, y_outlier)

train_acc_outlier = accuracy_score(y_outlier, clf_outlier.predict(
    X_outlier))
test_acc_outlier = accuracy_score(y_test, clf_outlier.predict(X_test))
    # Evaluate on the CLEAN test set

print("Training Accuracy with Outliers:", train_acc_outlier)
print("Test Accuracy with Outliers:", test_acc_outlier)
```

Summary

Logistic regression with L2 regularization performed extremely well on the banknote authentication dataset, achieving near-perfect separation. The model is sensitive to the regularization parameter λ and is negatively affected by outliers, which harm generalization performance.

Conclusion

This assignment demonstrated regression and classification using linear models. Key insights:

- Linear regression is effective but sensitive to feature scaling (for GD) and benefits from regularization.
- Ridge regression balances the bias-variance tradeoff and, when used on standardized data, allows coefficients to be interpreted as feature importances.
- Logistic regression with regularization improves classification robustness by controlling overfitting.
- Outliers harm generalization in both models, underlining the importance of data preprocessing and outlier detection.