Principles of Artificial Intelligence
Coursework

Antonio Di Stasio

Due 23/03/25

Contents

1 Preliminaries

1.1 Training Set vs Test Set

In machine learning, datasets are typically split into two distinct subsets: the **training set** and the **test set**. These serve different purposes in the model development process.

Training Set

- The **training set** is the portion of the dataset used to train the model.
- The model learns patterns and relationships in the data from the training set.
- It typically constitutes the majority of the dataset (e.g., 70-80%).
- The model's parameters (e.g., weights in a neural network) are optimized using this data.

Test Set

- The **test set** is the portion of the dataset used to evaluate the model's performance.
- It is kept separate from the training process and is only used after the model is fully trained.
- The test set simulates unseen data and helps assess how well the model generalizes to new, unseen examples.
- It typically constitutes a smaller portion of the dataset (e.g., 20-30%).

Key Difference

The key difference between the training set and the test set is their purpose:

- The **training set** is used to teach the model.
- The **test set** is used to evaluate the model's performance on unseen data.

Using a separate test set ensures that the model's performance metrics (e.g., accuracy) are not biased by the data it was trained on, providing a more realistic measure of its generalization ability.

1.2 Confusion Matrix and Accuracy

A **confusion matrix** is a table used to evaluate the performance of a classification model. For a binary classification problem, the confusion matrix is a 2×2 matrix:

$$\text{Confusion Matrix} = \begin{bmatrix} \text{True Positives (TP)} & \text{False Positives (FP)} \\ \text{False Negatives (FN)} & \text{True Negatives (TN)} \end{bmatrix}$$

Explanation of Terms

- **True Positives (TP):** The model correctly predicts the positive class.
- **False Positives (FP):** The model incorrectly predicts the positive class (actual is negative).
- **False Negatives (FN):** The model incorrectly predicts the negative class (actual is positive).
- **True Negatives (TN):** The model correctly predicts the negative class.

Accuracy

Accuracy is a metric that measures the proportion of correct predictions (both true positives and true negatives) out of all predictions. It is calculated as:

$$\text{Accuracy} = \frac{\text{TP} + \text{TN}}{\text{TP} + \text{TN} + \text{FP} + \text{FN}}$$

Example

Suppose you have the following confusion matrix for a binary classification problem:

$$\text{Confusion Matrix} = \begin{bmatrix} 50 & 10 \\ 5 & 35 \end{bmatrix}$$

Here:

- $\text{TP} = 50$
- $\text{FP} = 10$
- $\text{FN} = 5$
- $\text{TN} = 35$

The accuracy would be:

$$\text{Accuracy} = \frac{50 + 35}{50 + 35 + 10 + 5} = \frac{85}{100} = 0.85 \text{ or } 85\%$$

2 Linear Regression: Predicting California Housing Prices

In this exercise, you will build a linear regression model to predict median house values in California districts using the **California Housing Dataset**. You will explore the relationship between household income and house prices, evaluate the model's performance, and reflect on its limitations.

Dataset Description

The **California Housing Dataset** is a real-world dataset included in `scikit-learn`. It contains aggregated data from the 1990 U.S. Census, with the following structure:

- **Features**:
  - `MedInc`: Median household income in a district (scaled to $10,000s).
  - `HouseAge`: Median age of houses in a district.
  - `AveRooms`: Average number of rooms per household.
  - `AveBedrms`: Average number of bedrooms per household.
  - `Population`: Total population in a district.
  - `AveOccup`: Average number of household members.
  - `Latitude`: Latitude of the district.
  - `Longitude`: Longitude of the district.
- **Target**:
  - `MedHouseVal`: Median house value for households in a district (scaled to $100,000s).

Note: The dataset contains 20,640 samples. For simplicity, we'll focus on the `MedInc` feature to predict `MedHouseVal`.

Tasks

1. Load and Explore the Data

- Load the dataset using `sklearn.datasets.fetch_california_housing`.
- Plot a scatter plot of `MedInc` vs. `MedHouseVal` to visualize their relationship.
- Calculate summary statistics (mean, median, standard deviation) for both variables.

2. Preprocess the Data

- Split the data into training (80%) and testing (20%) sets using `train_test_split`.
- **Optional**: Standardize the feature using `StandardScaler` (note: linear regression coefficients are interpretable without standardization).

3. Build a Linear Regression Model

Train a linear regression model on the training data using both (Batch)[1] Gradient Descent and Stochastic Gradient Descent.

4. Make Predictions

- Predict house values for the test set.
- Predict the house value for a district with a median income of $80,000 (`MedInc = 8.0`).

6. Visualize the Results

- Plot the regression line overlaid on the test data scatter plot.
- Label axes appropriately and include a legend.

Discussion Points

- Why does income alone not fully explain house prices? Consider geographic factors (e.g., coastal vs. inland) and household size.
- Increase the number of features you think might improve the model and discuss it.

[1] Gradient Descent is also called Batch Gradient Descent as it works with the entire training set for updating the parameters.

3 Human Activity Recognition using Support Vector Machines

The goal of this exercise is to build and evaluate a binary classification model using Support Vector Machines (SVM) on the Human Activity Recognition (HAR) dataset from the UCI Machine Learning Repository. The original dataset contains sensor readings (accelerometer and gyroscope) from smartphones worn by volunteers performing six different daily activities (such as walking, standing, and sitting). However, you will simplify this multi-class problem by grouping the six activities into two classes—for instance, an "active" class (walking-related activities) and an "inactive" class (sitting, standing, or lying down). Specifically, you will have to

1. Load and preprocess the HAR data: Human Activity Recognition Using Smartphones Dataset
2. Convert the original 6-class labels into a binary (active vs. inactive) problem.
3. Train baseline SVM models with different kernels (linear, polynomial, RBF).
4. Perform hyperparameter tuning using `GridSearchCV`.
5. Evaluate and interpret results using confusion matrices and classification metrics.

In the following some steps for setting up the project.

3.1 Dataset Generation

- **Name:** Human Activity Recognition Using Smartphones
- **Activities (original):** `WALKING`, `WALKING_UPSTAIRS`, `WALKING_DOWNSTAIRS`, `SITTING`, `STANDING`, `LAYING`
- **Proposed Binary Labels:**
  - **Active (1):** WALKING, WALKING_UPSTAIRS, WALKING_DOWNSTAIRS
  - **Inactive (0):** SITTING, STANDING, LAYING
- **Features:** 561 features from accelerometer and gyroscope signals (time and frequency domain).
- **Size:** Over 10,000 samples (train + test combined).

Example Code: Below is a Python code snippet (in a single file) that you can adapt for the load and split of data. It shows how to:

1. Load and merge the data from the original train and test files.
2. Convert activities into binary labels.

Listing 1: Example Python code for binary SVM classification on HAR data

```
import pandas as pd
import numpy as np

# --- 1. LOADING THE DATA ---
# Adjust the paths to match where you've unzipped "UCI HAR Dataset"
```

```
PATH = "UCI␣HAR␣Dataset/"
features_path = PATH + "features.txt"
activity_labels_path = PATH + "activity_labels.txt"

X_train_path = PATH + "train/X_train.txt"
y_train_path = PATH + "train/y_train.txt"
X_test_path = PATH + "test/X_test.txt"
y_test_path = PATH + "test/y_test.txt"

# Load feature names
features_df = pd.read_csv(features_path, sep="\\s+", header=None, names=["
    idx", "feature"])
feature_names = features_df["feature"].tolist()

# Load activity labels (mapping IDs 1-6 to string names)
activity_labels_df = pd.read_csv(activity_labels_path, sep="\\s+", header=
    None, names=["id", "activity"])
activity_map = dict(zip(activity_labels_df.id, activity_labels_df.activity)
    )

# Load train/test sets
X_train = pd.read_csv(X_train_path, sep="\\s+", header=None, names=
    feature_names)
y_train = pd.read_csv(y_train_path, sep="\\s+", header=None, names=["
    Activity"])
X_test = pd.read_csv(X_test_path, sep="\\s+", header=None, names=
    feature_names)
y_test = pd.read_csv(y_test_path, sep="\\s+", header=None, names=["Activity
    "])

# Map the activity IDs to their names
y_train["Activity"] = y_train["Activity"].map(activity_map)
y_test["Activity"] = y_test["Activity"].map(activity_map)

# --- 2. CONVERT MULTI-CLASS TO BINARY ---
def to_binary_label(activity):
if activity in ["WALKING", "WALKING_UPSTAIRS", "WALKING_DOWNSTAIRS"]:
return 1 # Active
else:
return 0 # Inactive

y_train["Binary"] = y_train["Activity"].apply(to_binary_label)
y_test["Binary"] = y_test["Activity"].apply(to_binary_label)

# Now we have 0/1 labels in y_train["Binary"] and y_test["Binary"]
```

3.2 Reduction of the number of features

The HAR dataset has 561 features, which can be computationally heavy for large grid searches. One advanced approach is to include PCA (or another method like feature selection) in a pipeline. For example:

```
    from sklearn.pipeline import Pipeline
```

```
from sklearn.decomposition import PCA
from sklearn.svm import SVC
from sklearn.preprocessing import StandardScaler

pipeline = Pipeline([
('scaler', StandardScaler()),
('pca', PCA(n_components=50)), # reduce from 561 -> 50
('svc', SVC())
])
```

You can tweak $n_components$ (e.g., $30, 50, 100$) depending on how much dimension reduction you want vs. how much computational time you can spare.

3.3 Analysis and Extensions

If your new binary labels (Active vs. Inactive) are imbalanced, consider:

- Using `class_weight='balanced'` in `SVC()`.
- Evaluating `f1_score` or `precision/recall` instead of just accuracy.

3.4 Runtime Considerations

Polynomial kernels with large `degree` or large grids for `C` and `gamma` can be computationally expensive. Adjust `param_grid` or switch to `RandomizedSearchCV` if training takes too long.

3.5 Subject-wise Splits

For a more realistic scenario, ensure that no single subject appears in both train and test sets (i.e., do a *subject-wise* split). This tests generalization to *unseen* individuals.

3.6 Hyperparameter Tuning (GridSearchCV)

Given the high-dimensional data, carefully choose search spaces for C, γ, degree, etc. A wide search can be computationally heavy, but we'll outline the process. If it's too large, consider RandomizedSearchCV instead.

Example parameter grid:

```
from sklearn.model_selection import GridSearchCV

# Example pipeline (with optional PCA)
pipe = Pipeline([
('scaler', StandardScaler()),
# ('pca', PCA(n_components=50)), # uncomment if doing PCA
('svc', SVC())
])

param_grid = [
{
        'svc__kernel': ['linear'],
        'svc__C': [0.1, 1, 10, 100]
},
{
```

```
        'svc__kernel': ['poly'],
        'svc__C': [0.1, 1],
        'svc__degree': [2, 3],
        'svc__gamma': [0.001, 0.01, 0.1]
},
{
        'svc__kernel': ['rbf'],
        'svc__C': [0.1, 1, 10],
        'svc__gamma': [0.001, 0.01, 0.1]
}
]

grid_search = GridSearchCV(
estimator=pipe,
param_grid=param_grid,
scoring='accuracy', # or another multi-class metric, e.g., 'f1_micro'
cv=3, # might use 3-fold to save time (or 5-fold if feasible)
n_jobs=-1,
verbose=1
)

grid_search.fit(X_train, y_train.values.ravel())

print("Best parameters:", grid_search.best_params_)
print("Best cross-validation accuracy:", grid_search.best_score_)
```