MATH GR5430 Machine Learning for Finance
HomeWork 2

Minze Li
ml5163@columbia.edu

Sep 28, 2024

1 Overview of Supervised Learning

1.1 Definition and Key Concepts

Supervised learning is a fundamental paradigm in machine learning and statistics that aims to predict outputs from inputs. This seemingly simple definition encompasses profound mathematical and statistical principles. We can define supervised learning more rigorously through the following key assumptions:

1. **Data Generating Process (DGP):** The observed data pairs $(\mathbf{x}, \mathbf{y})$ are independent and identically distributed (i.i.d.) samples drawn from an unknown joint distribution $P(X, Y)$. This assumption is the foundation of statistical inference, allowing us to draw conclusions about the overall distribution from a finite sample.

2. **Conditional Independence:** Given the input X, the conditional distribution of the output Y can be expressed as:
$$Y = f(X) + \varepsilon$$
where $f(X)$ is the true underlying function we aim to learn, and ε is a random disturbance term. This term ε is assumed to have zero mean and a known distribution (often Gaussian), and its realizations are statistically independent under repeated sampling. This formulation allows us to separate the deterministic component $f(X)$ from the inherent randomness in the data.

3. **Functional Form:** The function $f : \mathbb{R}^d \to \mathbb{R}^k$ maps input space to output space. It is typically unknown, either partially or completely, but we assume it can be learned from observational data. The nature of this function - whether it's linear, nonlinear, smooth, or discontinuous - greatly influences the choice of learning algorithms.

4. **Finite-dimensional Parameter Space:** We often assume that f belongs to a parametric family governed by a finite-dimensional parameter vector $\theta \in \Theta \subseteq \mathbb{R}^p$. We can then write the function as $f(\mathbf{x}; \theta)$. This assumption allows us to reduce the problem of learning an infinite-dimensional function to estimating a finite number of parameters, making the learning process computationally tractable.

These assumptions form the theoretical foundation of supervised learning, guiding the development of algorithms and the analysis of their properties.

1.2 The Supervised Learning Process

The process of supervised learning involves several interconnected steps, each crucial for developing an effective predictive model:

1.2.1 Data Collection and Supervision

The first and often most challenging step is gathering a representative dataset:

$$D_{train} = \{(\mathbf{x}_i, \mathbf{y}_i) : i = 1, \ldots, n\}$$

This dataset, known as the training set, consists of input-output pairs. The term "supervised" in supervised learning comes from the presence of these output labels $\mathbf{y}_i$. The process of obtaining these labels, called annotation or labeling, can be time-consuming and expensive, especially for large datasets or when expert knowledge is required.

The quality and quantity of this data significantly impact the performance of the resulting model. Issues such as class imbalance, noise in the labels, or biased sampling can all lead to challenges in the learning process.

1.2.2 Model Selection

Choosing an appropriate model class to represent $f(X)$ is a critical decision in the learning process. This choice ranges from simple linear functions to complex neural networks, and it fundamentally shapes the capabilities and limitations of the resulting predictive system.

The model selection process involves navigating the bias-variance tradeoff:

- **Simple models** (e.g., linear regression) have low variance but potentially high bias. They're less likely to overfit the training data but may fail to capture complex patterns.
- **Complex models** (e.g., deep neural networks) have low bias but potentially high variance. They can capture intricate patterns in the data but are at risk of overfitting, especially with limited training data.

The goal is to find an optimal balance that minimizes overall error. This often involves techniques like cross-validation, where we evaluate model performance on held-out data to estimate its generalization ability.

1.2.3 Training

Once a model class is selected, the next step is to adjust its parameters to minimize the difference between its predictions and the actual outputs in the training set. Under the finite-dimensional parameter space assumption, this often involves solving an optimization problem:

$$\min_{\theta}[-\log p(\mathbf{y}|\mathbf{x}, \theta) + \log p(\mathbf{x})]$$

This formulation is derived from the principle of maximum likelihood estimation. The first term encourages the model to fit the observed data well, while the second term can be viewed as a form of regularization.

The joint likelihood of the entire training set is given by:

$$\prod_{i=1}^{n} p(\mathbf{y}_i|\mathbf{x}_i, \theta)p(\mathbf{x}_i)$$

Different optimization algorithms can be used to solve this problem, including gradient descent and its variants, which are particularly popular for training neural networks.

1.2.4 Evaluation and Prediction

After training, it's crucial to evaluate the model's performance on unseen data to assess its generalization ability. This is typically done using a separate test set or through cross-validation techniques.

The primary criterion for success in machine learning is often the quality of out-of-sample predictions. This focus on predictive performance distinguishes the machine learning approach from traditional statistical modeling, which may place more emphasis on parameter inference and model interpretability.

Various metrics can be used for evaluation, depending on the nature of the problem:

- For regression: Mean Squared Error (MSE), Root Mean Squared Error (RMSE), Mean Absolute Error (MAE)
- For classification: Accuracy, Precision, Recall, F1-score, Area Under the ROC Curve (AUC)

The choice of evaluation metric should align with the specific goals of the predictive task and the costs associated with different types of errors.

1.3 Common Approaches

1.3.1 Linear Models and Least Squares

Linear models form the foundation of many statistical learning techniques. They assume a linear relationship between the input features and the output:

$$Y = X^T\beta + \varepsilon$$

where β is a vector of coefficients to be learned, and ε is the error term.

The most common method for fitting linear models is least squares, which aims to minimize the Residual Sum of Squares (RSS):

$$RSS(\beta) = (\mathbf{y} - \mathbf{X}\beta)^T(\mathbf{y} - \mathbf{X}\beta)$$

If the matrix $\mathbf{X}^T\mathbf{X}$ is nonsingular, the closed-form solution for the Best Linear Unbiased Estimator (BLUE) is:

$$\hat{\beta} = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{y}$$

Linear models have several advantages:

- Simplicity and interpretability
- Computational efficiency
- Well-understood statistical properties

However, they also have limitations, particularly when dealing with nonlinear relationships in the data.

1.3.2 k-Nearest Neighbor Methods

k-NN is a non-parametric method used for both classification and regression. The k-NN model is defined as:

$$\hat{Y}(x) = \frac{1}{k}\sum_{x_i \in N_k(x)} y_i$$

where $N_k(x)$ is the neighborhood of x defined by the k closest training points.

Key characteristics of k-NN include:

- Simplicity: Easy to understand and implement
- Non-parametric nature: Makes minimal assumptions about the underlying data distribution
- Locality: Predictions are based on local information

An important consideration in k-NN is the choice of k. Smaller k values lead to more flexible models (low bias, high variance), while larger k values result in smoother decision boundaries (higher bias, lower variance).

Note: We cannot use training error to select k, as it would always favor $k = 1$, leading to overfitting. Instead, techniques like cross-validation are used to choose an appropriate value for k.

1.4 Structured Regression Models

While methods like k-NN and linear regression are foundational, they face challenges in certain scenarios:

- **Curse of Dimensionality:** In high-dimensional spaces, the concept of "nearest" neighbors becomes less meaningful. The volume of the space increases exponentially with the number of dimensions, causing data points to become sparse. This affects the performance of methods that rely on local neighborhoods, like k-NN.
- **Inefficient Use of Structure:** In low-dimensional settings, more structured approaches can often make better use of patterns in the data. For instance, if the underlying function is smooth, methods that exploit this smoothness (like spline regression) can outperform simple nearest-neighbor approaches.

1.5 Model Complexity and Error Decomposition

Understanding the relationship between model complexity and prediction error is crucial in supervised learning. The expected prediction error can be decomposed as:

$$\text{EPE} = \text{Irreducible Error} + \text{Bias}^2 + \text{Variance}$$

- **Irreducible Error:** This represents the inherent noise in the problem, which cannot be eliminated regardless of the model used.
- **Bias:** The bias term represents the error introduced by approximating a real-world problem with a simplified model. It's the difference between the expected (or average) prediction of our model and the correct value.
- **Variance:** This term measures how much the predictions for a given point vary between different realizations of the model.

This decomposition highlights the bias-variance tradeoff:

- Increased model complexity tends to decrease bias but increase variance. Complex models can capture intricate patterns in the training data but may fit noise, leading to poor generalization.
- Decreased complexity tends to increase bias but reduce variance. Simpler models are more stable but may miss important patterns in the data.

The goal in model selection is to find the sweet spot that minimizes total error. This is typically done through techniques like cross-validation, which provide estimates of out-of-sample performance for different model complexities.

1.6 Advanced Topics

1.6.1 Regularization and Penalized Methods

Regularization is a powerful technique for managing the bias-variance tradeoff. These approaches add a penalty term to the loss function:

$$\text{PRSS}(f; \lambda) = \text{RSS}(f) + \lambda J(f)$$

where $J(f)$ is a penalty that increases with the complexity of f, and λ is a tuning parameter that controls the strength of the regularization.

Common forms of regularization include:

- L1 regularization (Lasso): Encourages sparsity in the model parameters
- L2 regularization (Ridge): Shrinks all parameters towards zero
- Elastic Net: Combines L1 and L2 penalties

Regularization helps prevent overfitting by imposing a cost on complex models, effectively constraining the model space.

1.6.2 Basis Function Methods

These models use expansions of the form:

$$f_\theta(x) = \sum_{m=1}^{M} \theta_m h_m(x)$$

where $h_m(x)$ are basis functions. This approach allows for flexible nonlinear modeling while maintaining the computational benefits of linear methods. Common choices for basis functions include:

- Polynomials: Simple but can be unstable for high degrees
- Splines: Piecewise polynomials with smoothness constraints
- Radial Basis Functions: Particularly useful for interpolation in high-dimensional spaces
- Wavelets: Effective for capturing local features at different scales

The choice of basis functions and their number (M) allows for a flexible trade-off between bias and variance.

1.6.3 Kernel Methods

Kernel methods provide a powerful framework for nonlinear learning. They use kernel functions to implicitly map the input space to a high-dimensional feature space, where linear methods can then be applied. A common form of kernel-based estimation is:

$$\hat{f}(x_0) = \frac{\sum_{i=1}^{N} K_\lambda(x_0, x_i) y_i}{\sum_{i=1}^{N} K_\lambda(x_0, x_i)}$$

where K_λ is a kernel function that measures similarity between points.
Key advantages of kernel methods include:

- Ability to model complex nonlinear relationships
- Computational efficiency in high-dimensional spaces (kernel trick)
- Flexibility in defining notions of similarity between data points

Popular kernel methods include Support Vector Machines (SVM) for classification and Gaussian Process Regression for regression tasks.

These advanced methods provide powerful tools for tackling complex supervised learning problems, offering ways to balance model flexibility with generalization performance. They form the backbone of many state-of-the-art machine learning systems and continue to be areas of active research and development.

2 Pseudo-inverse Matrix Calculation and Validation

2.1 Computing the Moore-penrose Pseudo-inverse Matrix

The Singular Value Decomposition (SVD) can decompose a matrix A into the form $A = U\Sigma_S V^T$. The Moore-Penrose pseudo-inverse matrix is then given by $A^+ = V\Sigma_S^+ U^T$, where Σ_S^+ is obtained by taking the reciprocal of each non-zero element on the diagonal of Σ_S, while leaving the zeros unchanged. In numerical computations, only elements larger than a small tolerance are considered non-zero, with others replaced by zeros. And the small tolerance here is the numerical accuracy of NumPy.

```
# This is the Python Code of Homework 2
# @Author: Minze Li
# @Date: Sep 26, 2024
# MATH GR5430 Machine Learning for Finance
import numpy as np
from sklearn.linear_model import Ridge
```

```
# (a) Compute the Moore–Penrose pseudoinverse of a matrix using SVD.
def pseudoInverse(A):
    U, s, Vt = np.linalg.svd(A, full_matrices=False)
    s_inv = np.where(s > np.finfo(float).eps, 1/s, 0)
    return Vt.T @ np.diag(s_inv) @ U.T
```

2.2 Test by Generating a Ramdom Invertible Matrix

To validate our pseudo-inverse computation function, we generate random invertible matrices and compare their inverses with their computed pseudo-inverses. We conduct 1000 trials, recording the number of successful matches and calculating the success rate.

```
# (b) Test the pseudoinverse function by generating a random invertible square matrix.
def invertibleTest(N=1000):
    success_cnt = 0
    for i in range(N):
        n = np.random.randint(2, 50)  # Random size between 2 and 50
        A = np.random.rand(n, n)
        if np.allclose(np.linalg.inv(A), pseudoInverse(A), atol=np.finfo(float).eps): # use the default numerical accuracy of NumPy
            success_cnt += 1
        else:
            print(f"Test {i+1} failed")

    print(f"Invertible matrix tests passed: {success_cnt}/{N}")
    print(f"Invertible matrix tests success rate: {success_cnt/N*100:.2f}%")
invertibleTest()
```

And here are the outputs of the function:

```
Invertible matrix tests passed: 1000/1000
Invertible matrix tests success rate: 100.00
```

The 100% success rate strongly suggests that our pseudo-inverse function is correct for invertible matrices. However, it's important to note that this test only covers invertible matrices, and further testing is needed for singular and rectangular matrices.

2.3 Test by Ridge Regression

For matrices with linear dependencies (i.e., singular matrices), the standard inverse np.linalg.inv() is not applicable. In such cases, we need an alternative method to validate our pseudo-inverse computation. Ridge regression provides such an alternative. Ridge regression introduces a regularization term $\lambda\mathbb{I}_n$, which transforms a singular matrix into a non-singular one. As λ approaches zero, the solution to ridge regression approaches the Moore-Penrose pseudo-inverse solution. Specifically, by performing ridge regression on $\mathbb{I}_n = A^T\beta$, we obtain a regression coefficient $\hat{\beta} = (A^TA + \lambda\mathbb{I}_n)^{-1}A^T$, which approximates the pseudo-inverse of A as λ approaches zero.

```
# (c) Test the pseudoinverse function against the limit of ridge regression
def ridgeTest(N=1000):
    success_cnt = 0
    for i in range(N):
        m, n = np.random.randint(5, 10, size=2)
        A = np.random.rand(m, n)
        A[:, -1] = A[:, 0] # Make last column linearly dependent

        A_psd_inv = pseudoInverse(A)
        lambda_reg = 1e-8
        ridge_inv = np.linalg.inv(A.T @ A + lambda_reg * np.eye(n)) @ A.T

        if np.allclose(A_psd_inv, ridge_inv, atol=1e-4, rtol=1e-4): # verify the approximate equality
            success_cnt += 1
        else:
            print(f"Test {i+1} failed")
```

```
17
18      print(f"Ridge regression tests passed: {success_cnt}/{N}")
19      print(f"Ridge regression tests success rate: {success_cnt/N*100:.2f}%")
20  ridgeTest()
```

In this test, we generate matrices with a known linear dependency (by setting the last column equal to the first). We then compare our pseudo-inverse computation with the ridge regression approximation, using a small value of λ (1e-8).

And here are the outputs of the example:

```
Test 23 failed
Test 36 failed
Test 75 failed
Test 132 failed
Test 136 failed
Test 175 failed
Test 186 failed
Test 206 failed
Test 220 failed
Test 246 failed
Test 249 failed
Test 257 failed
Test 285 failed
Test 287 failed
Test 288 failed
Test 294 failed
Test 336 failed
Test 342 failed
Test 345 failed
Test 421 failed
Test 475 failed
Test 486 failed
Test 519 failed
Test 542 failed
Test 550 failed
Test 565 failed
Test 572 failed
Test 576 failed
Test 579 failed
Test 601 failed
Test 606 failed
Test 621 failed
Test 650 failed
Test 660 failed
Test 665 failed
Test 672 failed
Test 682 failed
Test 756 failed
Test 801 failed
Test 804 failed
Test 837 failed
Test 842 failed
Test 890 failed
Test 905 failed
Test 938 failed
Test 939 failed
Test 968 failed
Test 1000 failed
Ridge regression tests passed: 952/1000
Ridge regression tests success rate: 95.20%
```

The success rate of 95.20% indicates a strong agreement between our pseudo-inverse computation and the ridge regression approximation. The small discrepancy can be attributed to numerical precision issues and the fact that we're using a non-zero (albeit very small) value for λ.

This high success rate, combined with the perfect score on invertible matrices, provides strong evidence that our pseudo-inverse function is correctly implemented and numerically stable across a wide range of matrix types.

It's worth noting that the failed tests don't necessarily indicate errors in our pseudo-inverse computation. They could be due to numerical instabilities in the ridge regression approximation, especially

for ill-conditioned matrices. Further investigation of these specific cases could provide insights into the limitations and edge cases of both methods.

Also, I tried to use the ridge regression model in sklearn. In this regression, we use the original matrix A, which is an $n \times n$ square matrix. Then, we establish the regression model using the same $\lambda = 1e - 8$, and the resulting coefficient matrix is our computed pseudo-inverse.

```
# (c) Test the pseudoinverse function against the limit of ridge regression
def ridgeTest(N=1000):
    success_cnt = 0
    for i in range(N):
        n = np.random.randint(5, 10)
        A = np.random.rand(n, n)
        A[:, -1] = A[:, 0] # Make last column linearly dependent

        A_psd_inv = pseudoInverse(A)
        ridge = Ridge(alpha=1e-8, fit_intercept=False)
        ridge.fit(A, np.eye(n))
        ridge_coef = ridge.coef_.T

        if np.allclose(A_psd_inv, ridge_coef, atol=1e-4, rtol=1e-4): # verify the
    approximate equality
            success_cnt += 1
        else:
            print(f"Test {i+1} failed")

    print(f"Ridge regression tests passed: {success_cnt}/{N}")
    print(f"Ridge regression tests success rate: {success_cnt/N*100:.2f}%")
ridgeTest()
```

The outputs of this method are as follows:

```
Test 43 failed
Test 56 failed
Test 100 failed
Test 101 failed
Test 211 failed
Test 214 failed
Test 218 failed
Test 223 failed
Test 274 failed
Test 335 failed
Test 339 failed
Test 383 failed
Test 394 failed
Test 405 failed
Test 415 failed
Test 438 failed
Test 485 failed
Test 512 failed
Test 523 failed
Test 539 failed
Test 541 failed
Test 544 failed
Test 547 failed
Test 564 failed
Test 568 failed
Test 574 failed
Test 587 failed
Test 636 failed
Test 649 failed
Test 659 failed
Test 663 failed
Test 699 failed
Test 702 failed
Test 727 failed
Test 751 failed
Test 752 failed
Test 766 failed
Test 777 failed
Test 786 failed
Test 787 failed
```

```
Test 816 failed
Test 819 failed
Test 858 failed
Test 872 failed
Test 883 failed
Test 908 failed
Test 911 failed
Test 918 failed
Test 931 failed
Test 933 failed
Test 942 failed
Test 955 failed
Test 989 failed
Ridge regression tests passed: 947/1000
Ridge regression tests success rate: 94.70%
```

The success rate using this method is 94.70%, so our conclusion still holds.

In summary, we have verified that the inverse matrix of singular matrices obtained through ridge regression is almost completely consistent with the pseudo-inverse matrix.