Machine Learning Homework 1

April 2, 2025

2023141460251 Haoxiang Sun

Question 1: What is classification? How to perform classification by human? And what is the simplest way?

Classification is the task of assigning a category or label to an input based on its features. Mathematically, given an input $x \in \mathbb{R}^d$ (a d-dimensional feature vector) and a set of possible labels $Y = \{y_1, y_2, \ldots, y_C\}$ (where C is the number of classes), classification seeks a function $f : \mathbb{R}^d \to Y$ that maps x to a label y. This function can be deterministic (e.g., rule-based) or probabilistic (e.g., predicting class probabilities).

1.1 How humans perform classification

- Humans rely on prior knowledge, experience, and pattern recognition to classify objects. For example, to distinguish between a "cat" and a "dog," a human observes features such as shape (e.g., ear shape), size (e.g., body length), and behavior (e.g., barking vs. meowing). These features are compared to mental templates or prototypes formed from past observations, a process akin to unsupervised clustering in machine learning.
- This involves intuitive feature extraction—humans subconsciously select relevant attributes (e.g., fur texture) and ignore irrelevant ones (e.g., background color). Decision-making then follows, often based on similarity (e.g., "this looks more like a dog") or learned rules (e.g., "dogs have floppy ears"). For instance, a child might learn to classify fruits by associating "red and round" with "apple."
- The process is adaptive; humans refine their classification skills over time through feedback (e.g., correcting "that's a wolf, not a dog"). This mirrors supervised learning, where labels improve the model.

1.2 Simplest way

- The simplest method is *rule-based classification*, which uses a single, explicit criterion. For example, to classify objects as "large" or "small," a human might define a threshold θ (e.g., 10 cm) and apply the rule: "If the size $s(x) > \theta$, then label as large; otherwise, small." Mathematically:
$$f(x) = \begin{cases} \text{large} & \text{if } s(x) > \theta, \\ \text{small} & \text{otherwise.} \end{cases} \tag{1}$$
- This method requires minimal cognitive effort and no complex computation, making it intuitive even for novices. For instance, classifying emails as "spam" or "not spam" could

use a rule like "if the word 'free' appears, label as spam." However, its simplicity limits its robustness—single-rule systems fail when features overlap (e.g., some small dogs and large cats).

- In practice, humans might refine this by adding more rules (e.g., "if $s > \theta$ and weight $w > \phi$, then large"), but the single-rule approach remains the baseline for simplicity.

Question 2: What problem of 1NN is addressed by kNN?

The 1-Nearest Neighbor (1NN) algorithm classifies a test point x by assigning it the label of its nearest training point based on a distance metric, typically Euclidean distance $d(x, x_i) = \sqrt{\sum_{j=1}^{d}(x_j - x_{i,j})^2}$, where x_i is a training point.

2.1 Problem of 1NN

- *Sensitivity to noise and outliers*: Since 1NN relies solely on the single closest neighbor, it is highly vulnerable to noisy or mislabeled data. For example, if a training set contains an outlier (e.g., a mislabeled "cat" point in a "dog" cluster), and it happens to be the nearest neighbor to a test point, 1NN will incorrectly classify the test point based on that outlier.

- This sensitivity arises because 1NN has no mechanism to average out errors—it's a "local" decision with no robustness. In high-dimensional spaces, where noise is more prevalent due to the curse of dimensionality, this problem worsens. For instance, in a dataset with 10 features, a single noisy feature value can significantly distort the distance.

- A practical example: In a 2D dataset of heights and weights, if a "tall" person is mislabeled as "short" and lies near a test point, 1NN will misclassify despite the majority of nearby points being correctly labeled.

2.2 How kNN addresses this

- k-Nearest Neighbors (kNN) mitigates this by considering the k nearest neighbors (e.g., $k = 5$) and using majority voting for classification. For a test point x, kNN identifies the set $N_k(x)$ of k nearest neighbors and assigns the label:

$$y = \arg\max_{y_j} \sum_{x_i \in N_k(x)} \mathbb{I}(y_i = y_j), \tag{2}$$

  where $\mathbb{I}(y_i = y_j) = 1$ if $y_i = y_j$ and 0 otherwise, and y_j represents a class label.

- By averaging over multiple neighbors, kNN reduces the impact of individual outliers. For example, if $k = 5$ and one neighbor is an outlier, the other four (assuming they're correctly labeled) can outvote it, leading to a more robust prediction.

- This smoothing effect is particularly effective in noisy datasets. In the height-weight example, if four of the five nearest neighbors are "tall" and one is a mislabeled "short," kNN correctly predicts "tall." The choice of k trades off between noise reduction (larger k) and overfitting to local structure (smaller k).

Question 3: How to (why) incorporate the distance into classical kNN? And what will be benefited from it?

3.1 Why incorporate distance

- In classical kNN, all k neighbors contribute equally to the majority vote, regardless of their distance from the test point x. This assumes uniform relevance, which is often unrealistic—closer neighbors are typically more similar to x and thus more indicative of its true label, especially in dense or unevenly distributed datasets.
- For example, in a dataset with clusters of varying density, a distant neighbor from a sparse region might belong to a different class than a closer neighbor from a dense region. Ignoring distance can dilute the influence of local structure, leading to suboptimal predictions.
- The motivation stems from the intuition that similarity decreases with distance (e.g., in Euclidean space, $d(x, x_i)$ reflects dissimilarity). In real-world scenarios like image classification, pixels closer to a test pixel are more likely to share its label than those farther away.

3.2 How to incorporate distance

- Use *weighted kNN*, where each neighbor's vote is weighted by a function of its distance. A common approach is inverse distance weighting:
$$w_i = \frac{1}{d(x, x_i) + \epsilon}, \tag{3}$$
where $\epsilon > 0$ (e.g., 10^{-6}) prevents division by zero if $d(x, x_i) = 0$. The predicted label is then:
$$y = \arg\max_{y_j} \sum_{x_i \in N_k(x)} w_i \mathbb{I}(y_i = y_j). \tag{4}$$
- Alternatives include Gaussian weighting, $w_i = e^{-d(x,x_i)^2/\sigma^2}$, where σ controls the decay rate. For example, if $k = 3$ with distances 1, 2, and 3, weights might be $1/1 = 1$, $1/2 = 0.5$, and $1/3 \approx 0.33$, giving more influence to the closest neighbor.
- Implementation involves computing distances, sorting to find $N_k(x)$, applying the weight function, and aggregating weighted votes. This adds computational cost but leverages spatial information more effectively.

3.3 Benefits

- *Improved accuracy*: Closer neighbors, being more similar, dominate the decision, aligning predictions with local data patterns. For instance, in a dataset with overlapping classes, weighting ensures the nearest cluster has more say.
- *Robustness*: Reduces the impact of distant neighbors that might belong to different classes or clusters, especially in datasets with non-uniform density. In the image example, distant pixels with different lighting conditions are downweighted.
- *Flexibility*: Allows kNN to adapt to complex data distributions, bridging the gap between local (1NN) and global (large k) methods. Empirical studies often show weighted kNN outperforming classical kNN on noisy or imbalanced datasets.

Question 4: How to solve the scaling issue faced by kNN?

4.1 Scaling issue in kNN

- kNN relies on distance metrics like Euclidean distance, $d(x, x_i) = \sqrt{\sum_{j=1}^{d}(x_j - x_{i,j})^2}$, but features with different scales (e.g., height in meters [0-2] vs. weight in kilograms [0-150]) can dominate the distance calculation. For example, a 1-unit difference in weight (1 kg) contributes far more to the squared sum than a 1-unit difference in height (1 m).
- This imbalance skews results, as features with larger ranges or units overshadow others, even if they're less relevant. In a dataset with unscaled features like age (20-80 years) and income (10,000-100,000 USD), income differences will dominate, potentially misclassifying points based on income alone.

4.2 Solution

- *Feature normalization or standardization*:
  - *Normalization*: Scales features to a fixed range, typically $[0, 1]$:

$$x'_i = \frac{x_i - \min(x_i)}{\max(x_i) - \min(x_i)}, \tag{5}$$

    where $\min(x_i)$ and $\max(x_i)$ are computed over the training set. For age (20-80), 50 becomes $(50 - 20)/(80 - 20) = 0.5$.
  - *Standardization*: Transforms features to have zero mean and unit variance:

$$x'_i = \frac{x_i - \mu_i}{\sigma_i}, \tag{6}$$

    where $\mu_i = \frac{1}{n}\sum_{j=1}^{n} x_{j,i}$ and $\sigma_i = \sqrt{\frac{1}{n}\sum_{j=1}^{n}(x_{j,i} - \mu_i)^2}$. For age with $\mu = 50$, $\sigma = 15$, 65 becomes $(65 - 50)/15 = 1$.
- Apply scaling to both training and test data using training set statistics to avoid data leakage. Normalization is preferred when features have bounded ranges, while standardization suits Gaussian-like distributions or when preserving relative distances is key.
- Example: In a height-weight dataset, unscaled distances might prioritize weight; after standardization, both contribute equally, reflecting their true importance.

4.3 Benefit

- Ensures all features contribute equally to the distance metric, preventing bias toward large-scale features and improving classification fairness. In the age-income example, post-scaling, a 1-standard-deviation change in age has the same impact as a 1-standard-deviation change in income.
- Enhances accuracy by aligning distances with feature relevance, especially in high-dimensional spaces where unscaled features amplify noise. Studies show scaling often boosts kNN performance by 5-10% on real datasets.

Question 5: How to evaluate the performance of a classifier?

5.1 Evaluation methods

- *Accuracy*: The proportion of correct predictions:

$$\text{Accuracy} = \frac{\text{Number of correct predictions}}{\text{Total number of predictions}} = \frac{TP + TN}{TP + TN + FP + FN}. \tag{7}$$

  Simple but misleading in imbalanced datasets (e.g., 95% negative class yields high accuracy by always predicting negative).

- *Confusion Matrix*: For binary classification, a 2x2 table with True Positives (TP, correct positive predictions), True Negatives (TN), False Positives (FP, Type I errors), and False Negatives (FN, Type II errors). For multi-class, it's a $C \times C$ matrix.

- *Precision, Recall, F1-Score*:

$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}}, \quad \text{Recall} = \frac{\text{TP}}{\text{TP} + \text{FN}}, \quad \text{F1} = 2 \cdot \frac{\text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}}. \tag{8}$$

  Precision measures prediction reliability (e.g., spam detection), recall measures coverage (e.g., disease detection), and F1 balances both.

- *ROC Curve and AUC*: Receiver Operating Characteristic (ROC) plots True Positive Rate (TPR = Recall) vs. False Positive Rate (FPR = FP/(FP+TN)) across thresholds. Area Under the Curve (AUC) ranges from 0 to 1, with 1 indicating perfect separation. Useful for probabilistic classifiers (e.g., kNN with distance-based probabilities).

5.2 Practical steps

- *Train-test split*: Divide data into training (e.g., 80%) and test sets (20%) to evaluate generalization. For a dataset of 100 samples, train on 80, test on 20. Ensures the model isn't overfitting to training data.

- *Cross-validation*: Use k-fold (e.g., 5-fold) to split data into k subsets, training on $k - 1$ and testing on 1, repeating k times. Average score:

$$\text{CV Score} = \frac{1}{k} \sum_{i=1}^{k} \text{Score}_i. \tag{9}$$

  Reduces variance compared to a single split. For 100 samples and $k = 5$, each fold has 20 test samples.

- *Stratification*: In imbalanced datasets, ensure splits preserve class ratios (e.g., 10% positive class in both train and test). Tools like scikit-learn's 'StratifiedKFold' automate this.

Question 6: What is model selection? How to solve this issue?

6.1 What is model selection

- Model selection is the process of choosing the best model and its hyperparameters from a set of candidates based on performance. For kNN, this means selecting k (e.g., 1, 3, 5) or even comparing kNN to alternatives like SVM or decision trees.

- It addresses the bias-variance trade-off: small k risks overfitting (high variance), large k risks underfitting (high bias). The goal is to optimize generalization to unseen data, not just training fit.
- Example: In a dataset with two classes, $k = 1$ might perfectly fit training noise, while $k = 15$ might oversmooth, missing local patterns. Model selection finds the sweet spot.

6.2 How to solve

- *Cross-validation*: Split data into k folds (e.g., 5), train on $k-1$ folds, and validate on the remaining fold, averaging performance:

$$\text{Score} = \frac{1}{k}\sum_{i=1}^{k} \text{Validation Score}_i. \tag{10}$$

  For $k = 5$, each model (e.g., kNN with $k = 3$) is tested 5 times, reducing overfitting bias.
- *Grid Search*: Define a hyperparameter grid (e.g., $k = \{1, 3, 5, 7\}$, weights = {uniform, distance}) and evaluate all combinations via cross-validation. For 8 combinations and 5 folds, 40 models are trained. Select the best-performing set (e.g., $k = 5$, distance weights).
- *Information Criteria*: For parametric models (less common for kNN), use Akaike Information Criterion (AIC) or Bayesian Information Criterion (BIC) to balance fit and complexity. For kNN, heuristic metrics like silhouette score (clustering quality) could approximate this.

```python
from sklearn.neighbors import KNeighborsClassifier
from sklearn.model_selection import GridSearchCV
import numpy as np

# Sample data
X = np.random.rand(100, 5)
y = np.random.randint(0, 2, 100)

# Define model and parameter grid
knn = KNeighborsClassifier()
param_grid = {
    'n_neighbors': [1, 3, 5, 7],
    'weights': ['uniform', 'distance']
}

# Grid search with cross-validation
grid_search = GridSearchCV(knn, param_grid, cv=5, scoring='accuracy')
grid_search.fit(X, y)

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