CptS 570 Machine Learning

Homework - 3

Submitted by
Athul Jose P
11867566

School of Electrical Engineering and Computer Science
Washington State University

Contents

1 Q1

1.1 Q1.a

Let

$$E(x) = \frac{1}{d}\sum_{i=1}^{d} x_i, \quad E(z) = \frac{1}{d}\sum_{i=1}^{d} z_i$$

left side

$$\left(\frac{1}{\sqrt{d}}\sum_{i=1}^{d} x_i - \frac{1}{\sqrt{d}}\sum_{i=1}^{d} z_i\right)^2 = d \cdot (E(x) - E(z))^2$$

For the Euclidean distance, expand as:

$$\sum_{i=1}^{d}(x_i - z_i)^2 = \sum_{i=1}^{d} x_i^2 - 2\sum_{i=1}^{d} x_i z_i + \sum_{i=1}^{d} z_i^2$$

By Jensen's inequality and Chebyshev's inequality:

$$E(x^2) \geq (E(x))^2 \quad \text{and} \quad \frac{1}{d}\sum_{i=1}^{d} x_i z_i \geq \left(\frac{1}{d}\sum_{i=1}^{d} x_i\right)\left(\frac{1}{d}\sum_{i=1}^{d} z_i\right)$$

$$\sum_{i=1}^{d}(x_i - z_i)^2 \geq d \cdot (E(x) - E(z))^2$$

$$\left(\frac{1}{\sqrt{d}}\sum_{i=1}^{d} x_i - \frac{1}{\sqrt{d}}\sum_{i=1}^{d} z_i\right)^2 \leq \sum_{i=1}^{d}(x_i - z_i)^2$$

1.2 Q1.b

In high-dimensional spaces, computing Euclidean distances is computationally expensive. The left-hand side of the inequality, which uses the mean of the data points instead of individual distances, can serve as a computationally efficient similarity measure. By using:

$$\left(\frac{1}{\sqrt{d}}\sum_{i=1}^{d} x_i - \frac{1}{\sqrt{d}}\sum_{i=1}^{d} z_i\right)^2$$

simplify distance computations for nearest neighbor classification. Since the mean compresses the data into a single point, this approach reduces the computational cost while still maintaining a measure of similarity that is bounded by the Euclidean distance.

2 Q2 Summary

The article first reviews a family of nearest neighbor algorithms based on the concept of locality-sensitive hashing (LSH) and describes a hashing-based algorithm for the case where objects are points in a d-dimensional Euclidean space. The performance of this algorithm is provably near-optimal within the class of locality-sensitive hashing algorithms.

The nearest neighbor problem aims to find the data point closest to a query point within a dataset of n points. This is achieved by using a distance measure to assess the similarity of objects, where each object is represented as a point in $\mathbb{R}^d$. Examples of distance measures include the Manhattan distance, Euclidean norm, Jaccard coefficient, and Hamming distance. The primary task is to perform indexing or similarity searching for query objects. For low dimensions d, kd-trees are popular data structures used for multidimensional search. However, solutions for higher-dimensional data structures face limitations in space or query time that grows exponentially with d, often providing little improvement over a linear time search. To overcome this bottleneck, approximation methods have been proposed, where the algorithm returns a point whose distance from the query is at most c times the distance to the nearest point.

The key idea of LSH-based approximation in high dimensions is to hash the points using several hash functions, ensuring a higher probability of collision for points that are close to each other than for those that are far apart. In this article, the authors define a point p as an R-neighbor of a point q if the distance between p and q is at most R. The algorithm either returns one of the R-near neighbors or concludes that no such point exists for a given parameter R. Data structures are queried in increasing order of R, stopping when an answer is found.

The article defines two approximate near neighbor problems:

1. Randomized c-approximate R-near neighbor, or (c, R)-NN: Given a set P of points in a d-dimensional space $\mathbb{R}^d$ and parameters $R > 0$, $\delta > 0$, the data structure reports a cR-near neighbor of any query point q in P with probability $1 - \delta$. Success probability can be increased by building and querying multiple instances of the data structure.

2. Randomized R-near neighbor reporting: Given a set P of points in $\mathbb{R}^d$ and parameters $R > 0$, $\delta > 0$, the data structure reports all R-near neighbors of any query point q in P with probability $1 - \delta$. This structure may return many points if a large fraction of the dataset is near the query point, making a priori bounds on the algorithm's running time challenging to determine.

A function h selected randomly from a family H is considered locality-sensitive if it satisfies:

- If $||p - q|| \leq R$, then $\Pr_H[h(q) = h(p)] \geq P_1$.
- If $||p - q|| \geq cR$, then $\Pr_H[h(q) = h(p)] \leq P_2$.

For LSH to be effective, the condition $P_1 > P_2$ must hold. To amplify the desired collision probabilities (since the difference between P_1 and P_2 may be small), the algorithm concatenates multiple hash functions. This amplification increases the selectivity of hash functions, ensuring closer points have a higher probability of collision. The running time exponent of the authors' algorithm is essentially optimal, up to a constant factor.

3 Q3

Yes, converting a set of rules $R = \{r_1, r_2, \ldots, r_k\}$, where each rule r_i corresponds to a leaf node, into an equivalent decision tree is possible. According to the RBDT-1 method, this process involves constructing a decision tree where each node represents a decision based on the attributes in the rules. The RBDT-1 algorithm optimizes the tree's structure using three criteria to minimize depth and enhance interpretability. First, the Attribute Effectiveness (AE) criterion prioritizes attributes with the fewest "don't care" values, reducing unnecessary nodes and improving efficiency by increasing the likelihood of reaching a leaf node quickly. If multiple attributes achieve the highest AE scores, the Attribute Autonomy (AA) criterion further refines selection by choosing attributes that reduce the number of nodes needed to reach a leaf, creating a more direct path to the decision. In cases where multiple attributes still tie under AE and AA, the Minimum Value Distribution (MVD) criterion breaks the tie by selecting the attribute with the fewest distinct values in the current rule set, thus reducing the complexity of subsequent branches. This structured approach ensures that each rule r_i maps to a unique path from the root to a leaf, making the decision tree equivalent to the original set of rules.

4 Q4 Summary

In this paper, the authors compare discriminative learning, represented by logistic regression, and generative learning, represented by naïve Bayes, to analyze their effectiveness in classification tasks. They find that discriminative models, such as logistic regression, tend to have a lower asymptotic error, meaning they achieve better accuracy as the number of training samples becomes large. However, generative models, such as naïve Bayes, may reach their (higher) asymptotic error much faster. This characteristic can make generative classifiers advantageous in scenarios where fewer training examples are available, as they converge to their best performance with fewer data.

Generative classifiers learn by first estimating the joint probability distribution $p(x, y)$ of the input x and the label y, then applying Bayes' rule to compute the conditional probability $p(y|x)$, and finally selecting the label y with the highest probability. Discriminative classifiers, in contrast, aim to directly learn the mapping from input x to the class label y without modeling the joint distribution, focusing instead on distinguishing boundaries between classes. This direct approach allows discriminative models to reduce their error rate over time, as they better approximate the decision boundary with more data.

Through experiments, the authors demonstrate that for a large number of training examples, discriminative models eventually outperform generative models due to their lower asymptotic error. However, because generative models require only $O(\log n)$ samples to converge to their asymptotic error, while discriminative models require $O(n)$ samples, naïve Bayes may initially outperform logistic regression with smaller datasets. As the sample size grows, logistic regression catches up and typically surpasses naïve Bayes in accuracy.

This finding underscores the trade-off between discriminative and generative classifiers: discriminative models like logistic regression achieve better performance with large datasets, while generative models like naïve Bayes converge faster, making them suitable for smaller datasets. The authors suggest that understanding the conditions under which each approach excels can enable researchers to design hybrid classifiers that combine the strengths of both methods. Such hybrid models could dynamically adjust to data availability, performing well across a broader range of scenarios.

5 Q5

5.1 Q5.a

If the training data satisfies the Naive Bayes assumption (i.e., features are independent given the class label), then there are two possible outcomes for classifier performance as the training data approaches infinity. When the Naive Bayes conditional independence assumption is valid, Naive Bayes may reach its asymptotic error faster and thus potentially perform better in terms of accuracy if it converges quickly. In contrast, Logistic Regression may approach a lower asymptotic error over time because it directly models the conditional probability $P(Y|X)$, which is generally more accurate. Consequently, it is not immediately clear which classifier will perform better, as each exhibits distinct asymptotic behaviors: Naive Bayes may achieve faster convergence, while Logistic Regression could achieve a lower asymptotic error.

5.2 Q5.b

If the independence assumption of Naive Bayes does not hold, Logistic Regression will produce better results as the training data approaches infinity. Naive Bayes tends to have greater bias and lower variance than Logistic Regression, but the higher bias leads to less accurate results in cases where the independence assumption is violated. Logistic Regression, being a discriminative model, does not rely on the independence assumption and thus achieves better performance as data size grows.

5.3 Q5.c

Yes, we can compute $P(X)$ from the learned parameters of a Naive Bayes classifier. In Naive Bayes, $P(Y|X)$ is calculated using Bayes' rule:

$$P(Y|X) = \frac{P(X|Y)P(Y)}{P(X)}$$

Since $P(Y|X)$ and $P(X|Y)$ are estimated from the parameters learned during training, $P(X)$ can be calculated by marginalizing over Y using the law of total probability.

5.4 Q5.d

No, we cannot compute $P(X)$ from the learned parameters of a Logistic Regression classifier. Logistic Regression directly models $P(Y|X)$ without estimating $P(X|Y)$ or $P(Y)$. Since Logistic Regression only provides the conditional probability $P(Y|X)$, it lacks the information required to compute $P(X)$ directly.

6 Q6 Summary

In this paper, the authors review various ensemble methods, including bagging and boosting, and explain why ensemble approaches often perform better than individual classifiers. Ensemble classifiers work by combining the decisions of multiple individual classifiers, typically through a voting mechanism, either weighted or unweighted, to make a final classification. Constructing effective ensemble classifiers has become a prominent research area in supervised learning due to their ability to significantly enhance accuracy over single classifiers.

A crucial insight is that the effectiveness of ensemble classifiers relies on the diversity and accuracy of the individual classifiers. For an ensemble to be accurate, each classifier must perform better than random guessing, and for it to be diverse, classifiers should make uncorrelated errors on new examples. When diverse classifiers with error rates below 0.5 are combined, the ensemble benefits from a reduction in overall error through majority voting, leveraging the strengths of each classifier while offsetting their weaknesses.

The authors identify three fundamental reasons why ensemble methods are so effective: statistical, computational, and representational. Statistically, when the training data is insufficient to identify a single optimal hypothesis, ensembles help reduce the risk of overfitting by averaging predictions from multiple classifiers, providing a more stable decision. Computationally, many learning algorithms rely on local search methods that may converge to suboptimal solutions due to local minima. Ensembles mitigate this by constructing classifiers from different starting points, increasing the chance of finding better approximations. Representationally, ensembles expand the hypothesis space by combining individual hypotheses, which can approximate more complex decision boundaries that a single classifier may not capture. This is particularly useful when the true function lies outside the hypothesis space of individual classifiers.

Several ensemble methods are highlighted, including Bayesian voting, bagging, and boosting. Bagging creates multiple versions of the training set through bootstrapping and trains a classifier on each to build a stable ensemble, which is effective for unstable algorithms like decision trees. ADABOOST, a popular boosting algorithm, iteratively adjusts the weights of training samples to focus on those misclassified in previous rounds, creating a sequence of classifiers that increasingly target harder cases. The paper concludes with comparisons, noting that ADABOOST generally provides the best results in terms of accuracy, while bagging and randomized trees offer comparable performance, particularly on large datasets or in noisy environments where ADABOOST may overfit.

Through experiments and comparisons, the paper demonstrates that ensemble methods effectively address the shortcomings of single classifiers and provide robust solutions across diverse learning scenarios. This work suggests that ensemble approaches, particularly ADABOOST, have become valuable tools for improving classification performance in machine learning applications.

7 Q7 Summary

In this paper, the authors investigate five approximate statistical tests to assess whether one learning algorithm significantly outperforms another, with a focus on controlling the probability of Type I error—incorrectly detecting a difference when none exists. The primary goal is to provide robust comparisons for algorithms in single-domain learning tasks, especially when data availability is limited, necessitating resampling techniques such as cross-validation or bootstrapping. The tests reviewed are:

7.1 McNemar's Test

This test uses the chi-squared statistic to evaluate differences in paired misclassification counts, making it a good choice when the training data and algorithmic randomness introduce minimal variation. McNemar's test compares misclassification counts between two classifiers on the same test set, examining cases where one classifier correctly classifies an instance that the other misclassifies and vice versa. It is not sensitive to changes in the training data, so it should only be used when training set variability is not a concern.

7.2 Difference of Proportions Test

This test compares the error rates of two algorithms on a common test set by treating the number of misclassifications as a binomial random variable. However, this test assumes that the probability of misclassification is independent across the two algorithms, which may not hold if they are evaluated on the same test data. As a result, this test does not account for variation in training data or algorithmic randomness, which can limit its effectiveness in machine learning tasks where these factors are significant.

7.3 Resampled Paired t-Test

Commonly used in machine learning, this test involves multiple trials where the dataset is randomly divided into training and test sets, with both algorithms trained and tested on the same splits. Each trial yields a paired difference in error rates, and the t-test assesses whether the mean difference is statistically significant. However, this approach can suffer from inflated Type I error rates due to overlap between training and test sets in different trials, leading to dependence in the differences that violates the assumption of independent, identically distributed errors.

7.4 k-Fold Cross-Validated Paired t-Test

In this variant, the dataset is split into k mutually exclusive folds, with each fold used once as a test set while the remaining $k - 1$ folds form the training set. This process provides k estimates of the error difference. While this method reduces overlap in test sets, the training sets still overlap, which may underestimate the variance in performance differences. This test provides a better estimate than the resampled paired t-test but may still be affected by dependencies due to shared data across folds.

7.5 5x2cv Paired t-Test

The 5x2cv test performs five iterations of 2-fold cross-validation, where each dataset split provides a unique training and test set for each algorithm. This structure allows for an estimate of the mean performance difference and its variance, with improved control over Type I error rates by minimizing the dependency between training and test sets. This test assumes normality and independence of the differences across folds, making it a robust choice when variation in training data and algorithm randomness is a concern. Due to its resilience against Type I error inflation and its better handling of training set variability, the 5x2cv test is recommended for reliable comparisons of algorithm performance.

In summary, the paper highlights the importance of controlling for various sources of variation—such as test set selection, training set selection, internal algorithm randomness, and random classification error—when comparing learning algorithms. Of the tests reviewed, the 5x2cv paired t-test is identified as the most reliable, offering a good balance of power and control over Type I error. This test is particularly advantageous for experimental research in machine learning, where accurate and statistically sound comparisons between algorithms are critical for validating improvements in performance.

8 Q8

8.1 Q8.a

A real-world example where a state-action reward function is more suitable is flying an RC helicopter. In this task, the helicopter's rewards depend on both its state (position, velocity, orientation) and the specific action taken (e.g., throttle, pitch, roll, yaw). For instance, actions that stabilize the helicopter in place may yield positive rewards, while actions leading to instability or drift might result in penalties. This state-action dependency allows for more precise control by distinguishing between effective and ineffective actions in each state.

8.2 Q8.b

To adapt the Finite-Horizon Value Iteration algorithm for a state-action reward function $R(s, a)$, the update equation at each iteration t becomes:

$$V_t(s) = \max_{a \in A(s)} \left(R(s, a) + \sum_{s'} P(s'|s, a) V_{t+1}(s') \right)$$

In the original state-only reward function, the reward $R(s)$ depends solely on the state s. For a state-action reward function, we replace $R(s)$ with $R(s, a)$, allowing the reward to vary based on the action taken in a given state. This modified equation evaluates the value of taking action a in state s, then transitioning to the next state s' with the probability $P(s'|s, a)$. The algorithm proceeds to find the optimal policy by maximizing this expected reward for each action at each state and time step.

8.3 Q8.c

To transform an MDP with a state-action reward function $R(s, a)$ into an equivalent MDP with a state-only reward function $R(s)$, we introduce "bookkeeping" states that capture both the original state and the last action taken. For each state-action pair (s, a), we define a new state (s, a) in the transformed MDP, where the reward $R((s, a))$ is assigned as $R(s, a)$, directly corresponding to the state-action reward in the original MDP. Transitions in this new MDP are defined so that if the agent is in state (s, a) and moves to s' according to the original transition probability $P(s'|s, a)$, then the transition is represented as moving from (s, a) to the new bookkeeping state (s', a'), where a' is the next action taken. This ensures that optimal policies in the state-only reward MDP correspond exactly to those in the original MDP, preserving optimality across both representations.

9 Q9

To construct an equivalent standard (first-order) MDP $M' = (S', A, T', R')$ for a given k-order MDP $M = (S, A, T, R)$, we define S', A, T', and R' such that the new MDP M' has an equivalent solution to M in terms of optimal policies. The construction of each component is as follows:

9.1 State Set S'

In the k-order MDP M, the transition probability T depends on the current state s and the previous $k-1$ states. To convert this dependency into a first-order MDP, we redefine each state in S' as a sequence of k states from M:

$$S' = \{(s_{k-1}, \dots, s_1, s) \mid s_{k-1}, \dots, s_1, s \in S\}$$

where each state in S' is a tuple representing the current state and the previous $k-1$ states. This way, each "state" in M' captures the necessary history of k states required for a k-order MDP.

9.2 Action Set A

The action set A remains the same in M' as in M, since actions are applied to states without modification.

9.3 Transition Function T'

In the equivalent first-order MDP M', the transition probability T' depends only on the current tuple state $(s_{k-1}, \dots, s_1, s)$ and action a. Specifically, the transition from a state tuple $(s_{k-1}, \dots, s_1, s)$ under action a to a new state tuple $(s, s_{k-2}, \dots, s')$ is defined by:

$$T'((s_{k-1}, \dots, s_1, s), a, (s, s_{k-2}, \dots, s')) = T(s_{k-1}, \dots, s_1, s, a, s')$$

where $T(s_{k-1}, \dots, s_1, s, a, s')$ is the original k-order transition function from M. This ensures that T' in M' reflects the transition dependencies on k states in M.

9.4 Reward Function R'

The reward function R' in the equivalent MDP M' is defined based on the reward of the current state in M. For any tuple state $(s_{k-1}, \dots, s_1, s) \in S'$, set $R'((s_{k-1}, \dots, s_1, s)) = R(s)$. This means the reward in M' depends only on the current state s (the last state in the tuple), consistent with the rewards in M.

This construction captures the necessary state history in M as a single "state" in M'. Since each state in M' encodes all k states of history required by M, the first-order MDP M' fully represents the dynamics of the original k-order MDP M. An optimal policy for M', which maps states in S' to actions in A, can directly correspond to an optimal policy in M, where each action depends on the current and previous $k-1$ states in the original process. Thus, M' is equivalent to M in terms of optimal policies.

10 Q10

10.1 State-Action Reward Function

When the reward function depends on both the state s and the action a, the Bellman optimality equation with discount factor β is:

$$V_k(s) = \max_a \left(R(s,a) + \beta \sum_{s'} T(s,a,s') V_{k-1}(s') \right)$$

Here, $V_k(s)$ represents the value of being in state s under the optimal policy, where the agent selects the action a that maximizes the expected sum of the immediate reward $R(s, a)$ and the discounted future rewards.

10.2 State-Action-Next State Reward Function

When the reward function depends on the state s, the action a, and the resulting state s', the Bellman optimality equation with discount factor β is:

$$V_k(s) = \max_a \sum_{s'} T(s, a, s') \left(R(s, a, s') + \beta V_{k-1}(s')\right)$$

In this formulation, the value $V_k(s)$ depends on the action a that maximizes the expected reward, where the reward for transitioning from s to s' after taking action a is directly incorporated into the expectation.

11 Q11

Given the trivially simple MDP with states $S = \{s_0, s_1\}$, action $A = \{a\}$, rewards $R(s_0) = 0$, $R(s_1) = 1$, and transitions $T(s_0, a, s_1) = 1$ and $T(s_1, a, s_1) = 1$, we analyze the system with two different discount factors, $\beta = 1$ and $\beta = 0.9$.

11.1 Discount Factor $\beta = 1$

For state s_0:

$$V_\pi(s_0) = R(s_0) + \beta \cdot T(s_0, a, s_1) \cdot V_\pi(s_1)$$

$$V_\pi(s_0) = 0 + 1 \cdot V_\pi(s_1)$$

$$V_\pi(s_0) = V_\pi(s_1)$$

For state s_1:

$$V_\pi(s_1) = R(s_1) + \beta \cdot T(s_1, a, s_1) \cdot V_\pi(s_1)$$

$$V_\pi(s_1) = 1 + 1 \cdot V_\pi(s_1)$$

$$0 = 1$$

This results in a mathematical contradiction, indicating that the equations cannot be solved. This issue arises because, with $\beta = 1$, the MDP accumulates rewards indefinitely in state s_1, leading to an unsolvable system.

11.2 Discount Factor $\beta = 0.9$

For state s_0:

$$V_\pi(s_0) = R(s_0) + \beta \cdot T(s_0, a, s_1) \cdot V_\pi(s_1)$$

$$V_\pi(s_0) = 0 + 0.9 \cdot V_\pi(s_1)$$

$$V_\pi(s_0) = 0.9 \cdot V_\pi(s_1)$$

For state s_1:

$$V_\pi(s_1) = R(s_1) + \beta \cdot T(s_1, a, s_1) \cdot V_\pi(s_1)$$

$$V_\pi(s_1) = 1 + 0.9 \cdot V_\pi(s_1)$$

$$V_\pi(s_1) - 0.9 \cdot V_\pi(s_1) = 1$$
$$0.1 \cdot V_\pi(s_1) = 1$$
$$V_\pi(s_1) = 10$$

Substituting $V_\pi(s_1) = 10$ into the equation for $V_\pi(s_0)$:

$$V_\pi(s_0) = 0.9 \cdot 10 = 9$$

Therefore,

$$V_\pi(s_0) = 9 \quad \text{and} \quad V_\pi(s_1) = 10$$

This solution is valid because the discount factor $\beta < 1$ ensures that the rewards are finite, allowing the system to converge.

12 Fortune Cookie Classifier

The fortune cookie classifier is implmented using Naive Bayes classifier and the following accuracies are observed.

Training Accuracy	Testing Accuracy
93.1677	77.2277

Table 1: Accuracies for Naive Bayes Classifier