Homework #3 CPTS-570 Machine Learning

Ricardo Rivero - WSU ID 011843796

November 10, 2024

Analytical part.

1. Nearest-neighbor and the Jensen's inequality.

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

First, let us define the Difference Sequence and random variable as:

$$a_i = x_i - z_i \quad \textbf{for} \quad i = 1, 2, \cdots, d \tag{1}$$

Then, consider the random variable A that takes the values a_i with equal probability, which follows a uniform probability distribution over $\{1, 2, \cdots, d\}$ since each a_i corresponds to an index i and there are d indices. We can then rewrite the left hand side as:

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

And rewrite the Left-hand Side (LHS) in terms of the expected value of A

$$\left(\frac{1}{\sqrt{d}}\sum_{i=1}^{d} a_i\right)^2 = \left(\sqrt{d}\cdot\frac{1}{d}\sum_{i=1}^{d} a_i\right)^2 = \left(\sqrt{d}\cdot\mathbb{E}[A]\right)^2 = d\left(\mathbb{E}[A]\right)^2 \tag{3}$$

Then, we can express the Right-Hand side (RHS) as the sum of the squares of a_i:

$$\sum_{i=1}^{d} a_i^2 = d\cdot\frac{1}{d}\sum_{i=1}^{d} a_i^2 = d\cdot\mathbb{E}[A^2] \tag{4}$$

Now, we apply Jensen's inequality, which states that for a convex function f and a random variable X, $f\left(\mathbb{E}[X]\right) \leq \mathbb{E}[f(X)]$.
And since $f(x) = x^2$ is a convex function, we have:

$$\left(\mathbb{E}[A]\right)^2 \leq \mathbb{E}[A^2] \tag{5}$$

Which multiplying by d for both sides:

$$d\left(\mathbb{E}[A]\right)^2 \leq d\cdot\mathbb{E}[A^2] \tag{6}$$

Corresponds to $LHS \leq RHS$, therefore,

$$\left(\frac{1}{\sqrt{d}}\sum_{i=1}^{d} x_i - \frac{1}{\sqrt{d}}\sum_{i=1}^{d} z_i\right)^2 \leq \sum_{i=1}^{d}(x_i - z_i)^2 \tag{7}$$

1b. Using the proved property to enhance the computational efficiency of N.N computation.

To enhance the computation efficiency of the nearest neighbor using the above property we must understand both sides of the inequality as follows:

- Left-Hand side: square of the scaled mean of the component-wise (or dimension-wise) differences.
- Right-Hand side: The sum of squared Euclidean distances.

Given that the computation of exact Euclidean distances between all pairs of points becomes computationally intensive in high-dimensional spaces, we can leverage the inequality, such that the scaled mean difference is used as an approximate distance metric.

Let's assume that we have a static dataset, for which we can compute $\sum_{i=1}^{d} x_i$ for each point x once and store it in memory. Then we can query the dataset by computing the mean difference between the query point and candidate points such that for each point z we only calculate:

$$\frac{1}{\sqrt{d}} \sum_{i=1}^{d} (x_i - z_i) \tag{8}$$

Then, we can use this approximate distance to filter out the points that are unlikely to be the neareast neighbors. Since the approximate distance is always less than or equal to the Euclidean distance, if the approximate distance is large, the Euclidean distance must be at least as large.

2. Andoni et al about Locality Sensitive Hashing (LSH).

Andoni et al propose a new Locality Sensitive Hashing family, named Near-Optimal LSH Functions for Euclidean Distance. Their function is important because it allows for a faster query time, with query time exponent $p(c)$ that tend to $1/c^2$. Their has function has a starting point on the line partitioning method, where a point p is mapped into $\mathbb{R}^{\mathbb{I}}$ is partitioned into intervals of length w, which is a parameter. Then, the hash function for p returns the index of the interval containing the projection of p.

To leverage this method for bigger values of w, they utilized a "multi-dimensional version", by performing random projection into $\mathbb{R}^{\approx}$ where t is super-constant but of relatively small value. Then, instead of using grid partitioning, the do ball partitioning, which corresponds to creating a sequence of balls $B_1, B_2, ...$, with radius w and centers chosen independently at random.

They conclude that their new family of LSH functions yields a query time exponent of $p(c) = 1/c^2 + O(loglogn/log^1/3n)$, significantly improving the running time up to a constant factor.

3. Converting a set of rules into a decision tree.

Yes, a set of rules can be converted into a decision tree. For this, I will base my answer in the paper A Rules-to-Trees Conversion in the Inductive Database System VINLEN by Szydlo et al `https://doi.org/10.1007/3-540-32392-9_60` where they explain the AQDT-2 algorithm to build a decision tree from a set of attributional rules instead of examples.

In this algorithm, the input is a family of attributional rulesets (in this case, let's assume it's R), and the output is a decision tree. Then, we do the following

1. Evaluate each attribute occurring in the ruleset using the LEF attribute ranking measure and select the highest ranked attribute. Where LEF is $(\langle C, \tau_1, D, \tau_2, I, \tau_3, V, \tau_4, Do, \tau_5 \rangle)$.
2. Create a node of the tree, and assign to it the highest ranked attribute. Then create as many branches from the node, as there are legal values of the highest ranked attribute, and assign these values to the branches.
3. For each branch, associate a group of rules from the ruleset which contain a condition satisfied by the value assigned to this branch. And if there are rles in the ruleset context that do not contain the highest ranked attribute, add these rules to all rule groups associated with the branches.

4. If all the rules in a ruleset context for some branch belong to the same class, create a leaf node and assign to it that clas. If all the branches of the tree have class assignet, then stop. Else, repeat steps 1 to 4 for each beanch that has no class assigned.

4. Andrew Y. Ng et al: On Discriminative vs. Generative Classifiers: A comparison of logistic regression and Naive Bayes.

This study compares discriminative and generative classifiers, specifically logistic regression (LR) and naive Bayes (NB). The research demonstrates that while discriminative classifiers generally have lower asymptotic error, generative classifiers can approach their asymptotic error much faster. The analysis reveals two performance regimes as the training set size increases: initially, the generative model may outperform due to its faster approach to asymptotic error, but eventually, the discriminative model surpasses it by reaching a lower asymptotic error. Propositions are presented to support these findings, showing that the discriminative classifier's error is always less than or equal to the generative classifier's in the limit of infinite data.

Empirical results illustrate that Naive Bayes consistently reaches its asymptotic error faster than logistic regression. For discrete features, NB achieves lower error rates, while LR performs better with continuous features. Notably, for small sample sizes, logistic regression struggles to match the performance of Naive Bayes. These findings challenge the conventional wisdom that discriminative classifiers are always preferable, suggesting that the choice between generative and discriminative models should consider factors such as the nature of the features (discrete vs. continuous) and the available training set size.

5. Naive Bayes vs. Logistic Regression

5a. Infinite data that satisfies the Naive Bayes assumption.

When the training data satisfies the Naive Bayes assumptions and its size approaches infinity, **Logistic regression will produce better or equal results to Naive Bayes**. As stated in Proposition 1 in the Ng and Jordan paper, the asymptotic error of the discriminative classifier (Logistic regression in this case) is always less than or equal to that of the generative classifier (Naive Bayes). Adding on the empirical results obtained in the paper, Naive Bayes outperformed Logistic regression when the dataset was smaller. But when the data is larger, or approximates infinity, the Logistic Regression will at least match, and in some cases outperform NB by optimizing the conditional likelihood $p(y|x)$.

5b. Infinite data that does not satisfy the Naive Bayes assumption.

If the training data does not satisfy the Naive Bayes assumption, then the Naive Bayes' assumption will not align with the true join distribution $p(x, y)$, introducing bias into the model estimates and causing the classification error to not reach convergence. Instead, it will converge to a higher asymptotic error compared to the error that would be reached if the model was correctly specified (with conditional dependence given the class label). Contrastingly, the Logistic Regression model will remain correctly specified, leading to a correct estimation of the true parameters that define $p(y|x)$ as the training data size approaches infinity. Therefore, if the training data does not satisfy the Naive Bayes assumption, the Logistic Regression will produce better results compared to Naive Bayes.

5c. Computing $P(X)$ from the learned parameters of a Naive Bayes classifiers.

Yes, we can compute the marginal probability of the feature vector X using the parameters learned by Naive Bayes. Basically, the Naive Bayes learns:
1. The Class prior probabilities ($P(Y)$):

$$P(Y = y_k) = \frac{\text{Number of instances with class } \quad y_k}{\text{Total number of instances}}$$

2. The class-conditional feature probabilities ($P(X_j|Y)$, which under the Naive Bayes assumption are:

$$P(X|Y = y_k) = \prod_{j=1}^{n} P(X_j = x_j|Y = y_k)$$

Where n is the number of features. Then, we can compute $P(X)$ from learned parameters by computing the Joint Probability $P(X, Y)$ for each class:

$$P(X, Y = y_k) = P(Y = y_k) \cdot P(X|Y = y_k) = P(Y = y_k) \cdot \prod_{j=1}^{n} P(X_j = x_j|Y = y_k)$$

To finally, marginalize over all classes to obtain $P(X)$:

$$P(X) = \sum_{k=1}^{K} P(X, Y = y_k) = \sum_{k=1}^{K} P(Y = y_k) \cdot \prod_{j=1}^{n} P(X_j = x_j|Y = y_k)$$

Where K is the total number of classes.

5d. $P(X)$ cannot be computed from the learned parameters of a Logistic Regression.

No, we cannot estimate the marginal distribution of X $P(X)$ from the parameters learned of a Logistic Regression. Let us start by remembering that the Logistic Regression estimates $P(Y|X)$ using the sigmoid function:

$$P(Y = 1|X) = \frac{1}{1 + e^{-(\beta_0 + \beta_1 X_1 + ... + \beta_n X_n)}}$$

Where $\beta_0, \beta_1, ..., \beta_n$ are the learned parameters that represent the influence of each feature X_j on the log-odds of the classes, and they do not encode information about the distribution of X.

Therefore, even if we attempt to indirectly compute $P(X)$ by using the Bayes' theorem. Since the model does not provide $P(X|Y)$, $P(X)$ is incomputable, because:

$$P(X) = \sum_{Y} P(Y) \cdot P(X|Y)$$

6. Dietterich et al: Ensemble methods in Machine Learning

According to Dietterich et al, ensemble methods are learning algorithms that construct a set of classifiers and classify new data points by taking a weighted vote of their predictions and in their paper, they review Boosting and Bagging methods. They propose that one key to successful ensemble methods is to construct individual classifiers with error rates below 0.5, whose errors are somewhat uncorrelated. Then, they give three reasons why it is possible to construct very good ensembles:

1. **Statistical reason:** A learning algorithm can be viewed as searching a space H of hypotheses to identify the best hypothesis in the space. However, without sufficient data, the algorithm can find many different hypotheses that give the same accuracy on the training data, and by constructing an ensemble out of all these classifiers, the algorithms can be averaged in their votes and reduce the risk of choosing the wrong classifier.

2. **Computational reason:** An ensemble constructed by running a local search from many starting points may provide a better approximation to the true function than any individual classifier.

3. **Representational reason:** For most applications of machine-learning, the true function f cannot be represented by any of the hypotheses in H. Therefore, by forming weighted sums of hypotheses drawn from H, it may be possible to expand the space of representable functions.

Then, they review general purpose methods that can be applied to several machine-learning algorithms:

- **Bayesian Voting:** In this method, each hypothesis h in the hypothesis space H is assigned a posterior probability based on the training data S and prior beliefs $P(h)$. The prediction for a new data point x is made by aggregating the conditional probabilities $P(f(x) = y|S, x)$ of all hypotheses, weighted by their respective posterior probabilities $P(h|S)$. This ensemble approach is theoretically optimal when the true function f is drawn from H according to $P(h)$. For small training samples, many hypotheses contribute significantly, effectively averaging out uncertainties about f . However, as the training sample size grows, typically only one hypothesis dominates, reducing the ensemble to a single model. In complex scenarios where enumerating all hypotheses is infeasible, approximation methods like Markov Chain Monte Carlo (MCMC) are employed to sample hypotheses according to their posterior probabilities. Despite its theoretical appeal, Bayesian voting faces practical challenges, particularly in accurately defining the prior $P(h)$ and managing computational and representational complications. Then, when the prior knowledge is not captured adequately or computational constraints are significant, other ensemble methods could outperform Bayesian voting, as the Bayesian approach does not inherently resolve these practical issues.

- **Manipulating the Training Examples:** The authors refer to this method as constructing ensemble models by altering the training dataset to generate multiple hypotheses. This approach is particularly effective for unstable learning algorithms (such as decision trees, neural networks, and rule-based systems) that exhibit significant variation with minor changes in the training data. The section details Bagging (Bootstrap Aggregating), which involves creating multiple bootstrap replicates of the original dataset by sampling with replacement and training separate models on each replicate to reduce variance and prevent overfitting. It also discusses alternative sampling methods like dividing the training data into disjoint subsets, akin to cross-validated committees. Additionally, the AdaBoost algorithm is introduced as a sophisticated technique that iteratively adjusts the weights of training examples, giving more importance those misclassified by previous models to build a stronger ensemble. AdaBoost ultimately combines the individual classifiers through a weighted vote based on their accuracy, effectively minimizing an exponential error function and maximizing the training margin.

- **Manipulating the Input Features:** Another technique for generating multiple classifiers is altering the set of input features provided to the learning algorithm. For instance, Cherkauer's project on identifying volcanoes on Venus employed an ensemble of neural networks, each trained on different subsets of input features and varying network sizes. These feature subsets were manually selected to group features derived from distinct image processing techniques, such as principal component analysis (PCA) and the fast Fourier transform (FFT). This approach enabled the ensemble classifier to achieve performance comparable to that of human experts. Conversely, Tumer and Ghosh applied a similar method to a sonar dataset but discovered that removing even a few input features significantly degraded the performance of individual classifiers, leading to a poorly performing ensemble. It is important to acknowledge that manipulating input features effectively relies on the presence of highly redundant features within the dataset.

- **Manipulating the Output Targets:** This technique, known as error-correcting output coding (ECOC), is useful for handling multi-class classification problems with a large number of classes. The method involves partitioning the original set of classes into different subsets (e.g., A and B) multiple times to generate binary classification tasks. Each partitioning creates a new binary label, and a separate classifier is trained on each modified dataset. During prediction, each classifier votes based on its binary decision, and the final class is determined by aggregating these votes, often using methods like Hamming distance to match the closest class codeword. This approach not only improves the performance of various algorithms, such as decision trees and neural networks, but also enhances robustness by leveraging error-correcting capabilities. Additionally, extensions like AdaBoost.OC combine AdaBoost with ECOC to achieve superior ensemble performance with simpler implementation. Furthermore, integrating ECOC with feature selection techniques has demonstrated significant improvements in classification tasks, highlighting the versatility and effectiveness of manipulating output targets in ensemble learning.

- **Injecting Randomness:** Here the approach is generating diverse ensemble classifiers by introducing randomness into the learning process. For neural networks, this is achieved by initializing weights

randomly, leading to varied classifiers upon multiple training runs, though studies have shown that techniques like cross-validated committees and Bagging often outperform simple random initialization. In decision trees, randomness is injected by randomly selecting among the top-ranked features at each internal node, which significantly enhances ensemble performance compared to single trees. Similarly, for rule-based algorithms like FOIL, introducing randomness in selecting candidate conditions improves ensemble accuracy across most tasks. Additionally, combining bootstrap sampling with input feature noise, such as adding Gaussian noise to neural network inputs, has been demonstrated to yield substantial performance gains in both synthetic and medical diagnosis datasets. The section also highlights Bayesian ensemble methods that utilize Markov Chain Monte Carlo (MCMC) techniques to introduce randomness, allowing hypotheses to receive votes proportional to their posterior probabilities.

7. Dietterich. Approximate Statistical Test for Comparing Supervised Classification Learning Algorithms.

In this paper, the author discusses five statistical tests:

1. **McNemar's test:** Basically, the test consists in training two classifiers in a learning set R, yielding classifiers $\hat{f}_A$ and $\hat{f}_B$, and building a contigency table that includes the following: The number of examples misclassified by both models, the number of examples misclassified by $\hat{f}_A$ but not by $\hat{f}_B$, the number of examples misclassified by $\hat{f}_B$ but not by $\hat{f}_A$, and the number of examples misclassified by neither classifier, hereonwards n_{00}, n_{01}, n_{10}, and n_{11}. Then, it compares the distribution of counts expected under the null hypothesis to the observed counts (like a χ^2 test for goodness-of-fit), if we reject the null hypothesis, the two algorithms have different performance when trained on the training set R.

2. **A test for the difference of two proportions:** This second test measures the difference between the error rate of algorithm A and algorithm B, under the asumption that when algorithm A classifies an example x from the test set, the probability of misclassification is p_A, and the number of misclassifications of n test examples is a binomial random variable with mean np_A and variance $p_A(1-p_A)n$. If n is large enough, the binomial distribution can be approximated to a normal distribution, transforming the test into a two-sided test, where we would reject the null hypothesis that the error are not different, if $|z| > Z_{0.975} = 1.96$

3. **The resampled paired t test:** In this test, a series of trials is conducted where the available data S is randomly divided into a training and a test set. Then both algorithms are trained on the training data and the resulting classifiers are tested on the test data. Under the assumption that the differences $p^{(i)})p_A^{(i)} - p_B^{(i)}$ were drawn independently from a normal distribution, a Student's t test can be applied.

4. **k-fold cross-validated paired t test**: Is basically the same test as the previous, but instead of randomly splitting the data into training and test, the data S is randomly divided into k disjoints sets of equal size. Then k trials are conducted, where the test set is T_i and the training set is the union of all the other training sets.

5. **5x2cv paired t test:** This test aims to overcome an issue with the k-fold cross-validated paired t test, where the t statistic was too large in some cases. This was caused by a poor estimation of the means, which can be related to correlations between different folds. However, the authors found that if the numerator of the t statistic is replaced with the observed difference from a single fold of the k-fold validation, the statistic "behaves" well. The key modification is that they perform 5 replications of 2-fold cross-validation.

Finally, they found that the 5x2cv test was the most powerful among the statistical tests that have acceptable Type I error. Also, the test was able to assess the effect of both the choice of training set, and the choice of test set.

8.Finite-Horizon MDPs

8a. Real world problem where the MDP is more naturally modeled using $R(s,a)$ than $R(s)$.

Some examples where including the action in the reward function could be autonomous driving, whether it is self-driving cars or autonomous drones for surveillance or delivery. In these examples, $R(s,a)$ would perform better because the agent can receive precise feedback on its maneuvers or decisions, where an action such as switching lanes without using the turn signal might incur a penalty due to risk of collision. Additionally, in this scenario, the state-action function would allow the agent to associate specific actions with their outcomes, whether they are positive or negative. In a scenario like autonomous vehicles, using $R(s,a)$ would lead to enhanced learning efficiency, allowing for faster convergence because the algorithm can quickly identify actions that lead to desirable states, thus accelerating the policy optimization process. Finally, all of these together would also decrease the number of iterations (and potentially of damaged cars/drones) required to refine the decision-making process.

8b. Modified Finite-horizon value iteration algorithm fir $R(s,a)$.

To incorporate action dependent rewards, we can modify the Finite-horizon value iteration for each state $s \in S$:

$$V_\pi^k(s) = \max_{a \in A} \left[R(s,a) + \beta \sum_{s\prime \in S} T(s,a,s\prime) V_\pi^{k-1}(s\prime) \right]$$

The key modification is that the original equation assumes a single policy π and does not explicitly considers different actions, where the new equation explicitly maximizes over all possible actions a in each state.

Additionally, in the modified version we include transition probabilities, understanding that actions often influence not just immediate rewards but also the likelihood of transitioning to subsequent states. Finally, we include a recursive dependency on previous horizon ($V_\pi^{k-1}(s')$), ensuring that the value function accurately reflects the expected cumulative reward over the remaining horizon. Therefore, the modified value iteration algorithm would be defined as:

Algorithm 1 Finite-Horizon Value Iteration with State-Action Rewards

Require: States S, Actions A, Transition Probabilities $T(s,a,s')$, Reward Function $R(s,a)$, Discount Factor β, Horizon K

Ensure: Value Function $V^k(s)$ for all $s \in S$ and $k = 0, 1, \ldots, K$

```
Initialize V^0(s) = 0 for all s ∈ S
for k = 1 to K do
    for each state s ∈ S do
        V^k(s) ← max_{a∈A} [R(s,a) + β Σ_{s'∈S} T(s,a,s') V^{k-1}(s')]
    end for
end for
return V^k(s) for all s ∈ S
```

8c. Any MDP with a state-action reward function can be transformed into an equivalent MDP with just state-reward function.

Briefly, an MDP with a state-action reward function can be transformed into an MDP with just a state-reward function by introducing nookkeeping states that represent the actions taken from each original state, as follows:

$$S\prime = S \cup \{s_a \mid s \in S, a \in A\}$$

This means that for every state $s \in S$ and action $a \in A$, we create a unique bookkeeping state s_a which represents the action a taken from state s.

We then define a new action space A' by introducing a single dummy action a' to simplify transitions from bookkeeping states.

$$A\prime = \{a \in A\} \cup \{a\prime\}$$

It is also necessary to define a new transition function T' fron original states, such that taking action a in state s deterministically transitions the agent to s_a. And from bookkeeping states, such that from s_a, taking the dummy action a' transitions the agent to the next state s' based on the original transition probabilities.

For the reward function, we define $R'(s) = 0$, meaning that no reward is assigned when the agent is in an original state, and from bookkeeping states $R'(s_a) = R(s, a)$.

Finally, we get a value function equivalence as follows:

$$V\prime^{(s)} = \max_{a \in A} \left[R(s, a) + \gamma \sum_{s\prime \in S} T(s, a, s\prime) V\prime^{(s\prime)} \right] = V*^{(s)}$$

9. k-th Order MDPs.

To transform a k-th order MDP M = (S, A, T, R) into an equivalent first-order MDP $M\prime = (S\prime, A\prime, T\prime, R\prime)$, we can follow these steps:

1. Define the new state space $S\prime$:
   - $S\prime$ consists of tuples that capture the history of the last k-1 states along with the current state.
   - Formally: $S\prime = S^k$, where each state in $S\prime$ is represented as $(s_{t-k+1}, s_{t-k+2}, \ldots, s_t)$, where s_t is the current state.
2. Define the action space $A\prime$:
   - $A\prime = A$
   - The action space remains unchanged from the original MDP.
3. Define the Transition function $T\prime$:
   - $T\prime$ determines the probability of transitioning to a new state tuple given the current state tuple and an action.
   - Formally: $T\prime((s_1, s_2, \ldots, s_k), a, (s_2, s_3, \ldots, s_{k+1}))) = T(s_k, a, s_{k+1})$, this means the next state tuple shifts forward by one, incorporating the new state s_{k+1} resulting from taking action a in state s_k.
4. Define the Reward function $R\prime$:
   - $R\prime$ assigns rewards based on the current state and the action taken, similar to the original MDP.
   - Formally: $R\prime((s_1, s_2, \ldots, s_k)), a) = R(s_k, a)$, the reward depends only on the current state s_k and the action a , aligning with the original reward structure.

By implementing an expansion of the state space to include the necessary history of previous states, the k-th order dependencies are included within the new state tuples, allowing the transformed MDP M' to retain the dynamics of the original k-th order MDP while fitting the framework of a standard first-order MDP.

10. Bellman optimality equation with discount factor β.

For the reward function $R(s, a)$, the reward depends only on the current state s and the action a taken by the agent. In this case, the Bellman optimality equation is given by:

$$V^{(s)} = \max_{a \in A} \left[R(s, a) + \beta \sum_{s\prime \in S} P(s\prime \mid s, a)\, V^{(s\prime)} \right]$$

For the reward function $R(s; a; s\prime)$, the reward depends on the current state s, the action taken a, and the resulting next state $s\prime$. Then, the Bellman optimality equation becomes:

$$V^{(s)} = \max_{a \in A} \left[\sum_{s\prime} P(s\prime \mid s, a) \left(R(s, a, s\prime) + \beta V^{(s\prime)} \right) \right]$$

11. Trivially simple MDP.

Briefly, the stated problem is the following:

- States: $S = \{s_0, s_1\}$
- Actions: $A = \{a\}$ (only one action available)
- Reward Function: $R(s_0) = 0$; $R(s_1) = 1$
- Transition Function: $T(s_0, a, s_1) = 1$ (from s_0 , action a leads deterministically to s_1); $T(s_1, a, s_1) = 1$ (from s_1 , action a keeps the agent in s_1)

11a. Linear equations for evaluating the policy and solving the linear system if $\beta = 1$

For the mentioned formulation, we can evaluate the policy by calculating the Bellman optimality equation for each state s:

$$V^{(s)} = R(s) + \beta \sum_{s\prime} T(s, a, s\prime) V^{(s\prime)}$$

Which given that $\beta = 1$, yields that for state s_0:

$$V^{(s_0)} = R(s_0) + \beta T(s_0, a, s_1) V^{(s_1)} = 0 + 1 \times 1 \times V^{(s_1)} = V^{(s_1)}$$

And for state s_1:

$$V^{(s_1)} = R(s_1) + \beta T(s_1, a, s_1) V^(s_1) = 1 + 1 \times 1 \times V^{(s_1)} = 1 + V^{(s_1)}$$

If we solve the system of linear equations for s_1 and substract $V(s_1)$ from both sides, we get that $0 = 1$, indicating that the system has no finite solution. This can be understood from the light of the future rewards having the same value of immediate rewards when $\beta = 1$

11b. Linear equations for evaluating the policy and solving the linear system if $\beta = 0.9$

Leveraging the reasoning from the previous question, if $\beta = 0.9$, if we solve for s_1 and subtract $0.9V(s_1)$ from both sides:

$$\begin{aligned} V^{(s_1)} - 0.9V^{(s_1)} &= 1 \\ 0.1V^{(s_1)} &= 1 \\ V^{(s_1)} &= \frac{1}{0.1} = 10 \end{aligned}$$

Then we can solve for $V(s_0)$, which yields that $V(s_0) = 9$. Therefore, the discount factor $\beta = 0.9$ ensures that rewards are weighted less than immediate rewards.

Programming and Empirical Analysis.

1. Fortune Cookie Classifier using Naive Bayes

We implemented a Naive Bayes classifier to classify fortune cookie messages into two classes: Class 0: Messages that contain a wise saying. Class 1: Messages that predict what will happen in the future.

To begin, we implemented a preprocessing function that takes the raw input file and iterates over the lines of the file, then splits the lines into words and removes the words that are found in the stoplist.txt file. Then, the remaining vocabulary is sorted alphabetically, and the data is converted into a set of features where each slot in the feature vector takes a value of 0 or 1.

Then, for the classifier, we implemented the Naive Bayes classifier with a Laplace Smoothing where the probability of a label is calculated by $P(label) = \frac{nlabel+1}{ntotalLabels+2)}$, and the probability of observing a word is given by $P(word) = \frac{wordCount+1}{Total_Words_in_class+VocabularySize}$.

For our implementation, we obtained a training and testing accuracy of 95.96% and 83.17%.

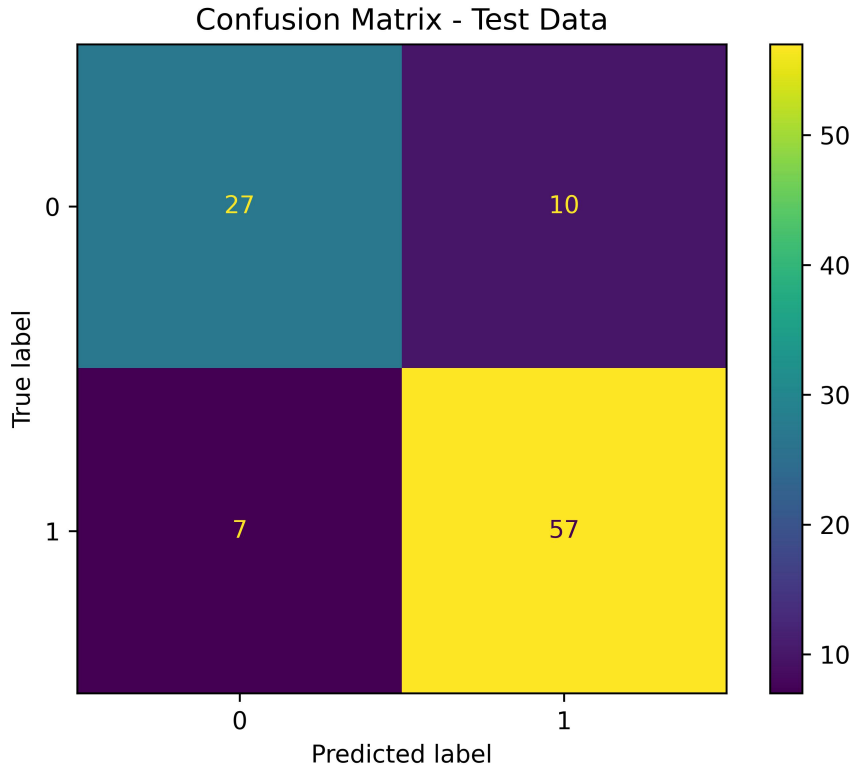

Figure 1: Confusion Matrix for our implementation of the Naive Bayes classifier on the testing data.

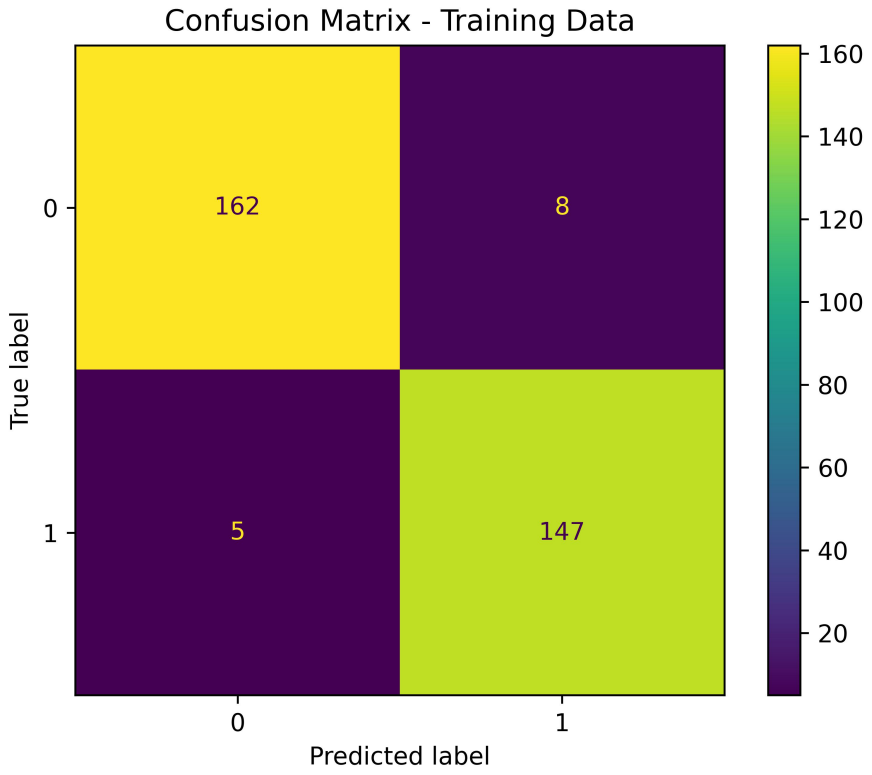

Figure 2: Confusion Matrix for our implementation of the Naive Bayes classifier on the training data.

We then validated our implementation against the Naive Bayes implementation available from Sci-kit Learn, and obtained the same training and testing accuracy (95.96% and 83.17%)

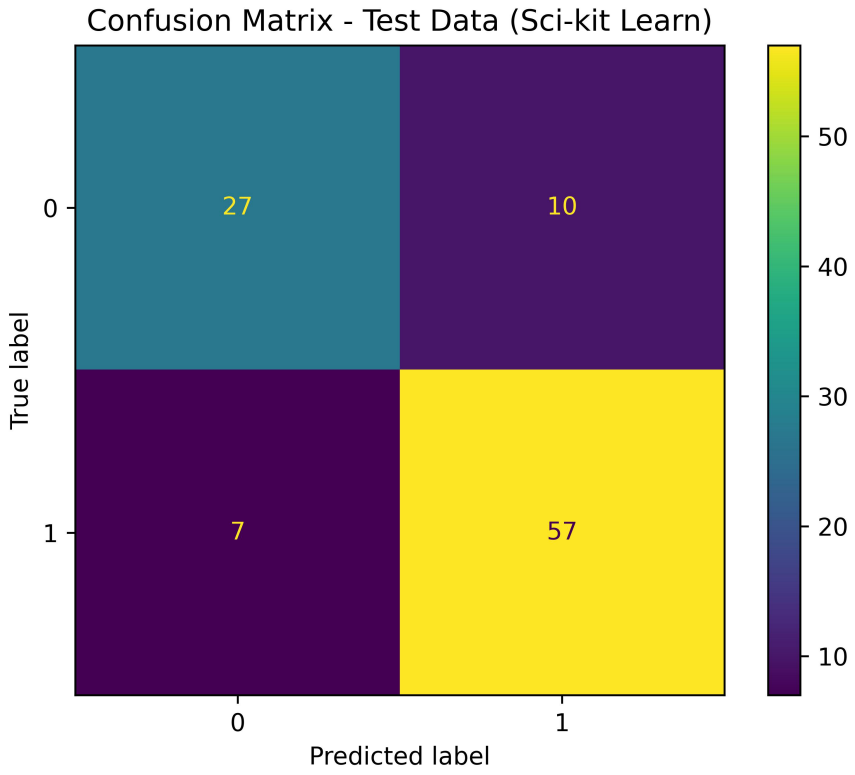

Figure 3: Confusion Matrix for Sci-kit Learn's implementation of the Naive Bayes classifier on the testing data.

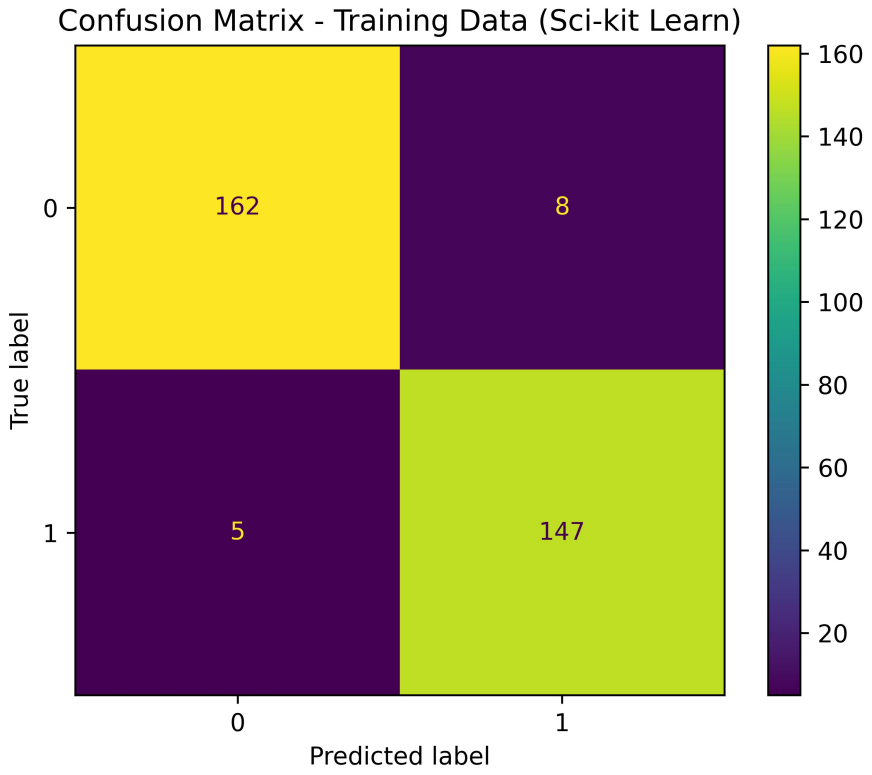

Figure 4: Confusion Matrix for Sci-kit Learn's implementation of the Naive Bayes classifier on the training data.