Final Individual Project

ShouTai Yue

DSC 148: Introduction to Data Mining
Professor JingBo Shang

14 March, 2025

Table of Contents:

Decision Tree

Overview: Decision trees belong to the family of supervised learning algorithms, where the model is trained on a labeled dataset. It is a tree-like model that learns decision rules that map input features to output values. Their appeal lies in their simplicity and interpretability. The structure, much like a flowchart, allows users to easily follow the "if-else" rules that lead to a prediction, making them useful for both explaining and diagnosing model behavior. The model is not as robust to noisy data, but they are effective in capturing non-linear trends, as well as the flexibility to handle varied data types and structures.

Implementation Details: There are various important formulas in the implementation of Decision Trees, the most import of which is the formula for the calculation of entropy as follows:

$$H(y) = -\sum_{i=1}^{K} p_i \log_2 p_i$$

Where P_i is the proportion of class i in the node (P_i = count of yi / total samples). This formula is the basis of how decision rules are formed, as entropy calculates the 'impurity' of a node. The gain from a split is calculated as the difference between the parent node's entropy and the weighted average entropy of the children:

$$\text{IG} = H(\text{parent}) - \left(\frac{N_{\text{left}}}{N} H(\text{left}) + \frac{N_{\text{right}}}{N} H(\text{right})\right)$$

It shows approximately how much improvement is being made with each iteration of the tree. The decision tree includes various functions to implement, including:

- **entropy(lst):** Computes the entropy of a label vector.
- **information_gain(lst, values, threshold):** Calculates the information gain of a candidate split using the entropy measure.
- **find_rules(data):** For each feature (each column in the dataset), unique values are extracted and candidate split thresholds (midpoints) are computed.
- **next_split(data, label):** Evaluates each candidate split across the selected features to choose the best split based on maximum information gain.
- **build_tree(X, y, depth):** Recursively builds the tree. The base cases include reaching the maximum depth (if specified), having too few samples to split, or the node being pure (all labels identical).
- **ind_predict(inp) and predict(inp):** Traverse the built tree for individual test samples and a batch of test samples, respectively.

Each of which is a key component in creating a decision tree, and will be used in the following steps in actually building the classifier

1. **Determine the Best Split:** Use next_split to choose the feature and threshold that best splits data
2. **Create a New Node:** Make a node that stores the chosen feature, threshold, and the current depth.
3. **Check for Stopping Conditions:** If the maximum depth is reached, there are too few samples, or no valid feature is found, assign the majority label to the node, mark it as a leaf, and stop.

4. **Check for Purity:** If all samples have the same label, mark the node as a leaf with that label and stop.
5. **Split the Data:** Divide the data into two groups: one where the feature value is less than or equal to the threshold, and one where it's greater.
6. **Handle Empty Splits:** If one group is empty, mark the node as a leaf using the majority label.
7. **Recursively Build Subtrees:** Apply the same process to the left and right subsets, increasing the depth by one for each recursive call.
8. **Return the Node:** Once both subtrees are built, attach them to the node and return it.

Takeaways: While implementing the model, the primary difficulty that was encountered was the different base cases in the implementation of next_split. There were a number of cases that I had failed to consider in the initial implementation, and later was fixed by repeatedly running the test cases for the code, and reading the error message.

Random Forest

Overview: The Random Forest classifier builds an ensemble of decision trees, each trained with random feature selection and candidate split constraints. The final prediction is determined by a majority vote (with random tie-breaking if necessary). In essence, it fits many decision trees to minimize bias and random error.

Implementation Details: The RandomForest class wraps the DecisionTree and creates an ensemble:

- fit(X, y): Trains multiple DecisionTree classifiers (as many as specified by n_trees). Each tree is built using the same parameters (such as max_depth, size_allowed, n_features, and n_split).
- ind_predict(inp): For a single test instance, it gathers predictions from all trees and selects the final prediction by majority vote. Ties are broken using a random selection.
- predict_all(inp): Applies ind_predict to each instance.

These methods will be used in the following steps to create and fit a decision tree:

1. **Build Multiple Trees:** For each tree in the forest, create a new Decision Tree using the specified parameters (e.g., maximum depth, minimum samples, number of features, candidate splits).
2. **Train Each Tree:** Fit each Decision Tree on the training data. (Optionally, you can use bootstrapping, though the provided implementation trains on the full dataset.)
3. **Store the Trees:** Add every trained tree to the forest ensemble.
4. **Make Predictions:** For a given test instance, obtain a prediction from each tree.
5. **Combine Predictions:** Determine the final prediction by applying majority voting to all tree predictions. If there's a tie, randomly select one of the tied classes.

Takeaway: Not much error or difficulties were encountered during implementation and running of the Random forest, it is simply a model that will create an ensemble of decision trees to reduce the effect of random noise at the cost of more run time and additional hyperparameters to tune.

Matrix Factorization

Overview: Matrix Factorization is a collaborative filtering technique used to predict user–item interactions by decomposing the original rating matrix into two lower-dimensional matrices. This unified model can operate in two modes: one that includes bias terms (global, user, and item biases) and one that does not.

- **Without Bias:** Focuses solely on learning user and item latent factors, using the dot product of these factors to predict a rating.
- **With Bias:** Adds additional terms to account for global (average) rating tendencies, as well as systematic user and item deviations. This typically leads to improved accuracy by capturing consistent behaviors (e.g., users who always rate high or items that tend to receive high ratings).

Both models are implemented to allow a more comprehensive exploration to see which approach better fits a given dataset.

Implementation Details: To start we explore the core formulas used in Matrix Factorization. The prediction rules are as follows:

- **Without Bias:** $\text{prediction} = P[u] \cdot Q[i]$ Where P[u] is the latent factor vector for user u, and Q[i] is the latent factor vector for item i.
- **With Bias:** $\text{prediction} = b + b_u[u] + b_i[i] + (P[u] \cdot Q[i])$ Where b is the global average, b_u[u] is the user-specified bias, b_i[i] is the item-specified bias, and P[u] dot Q[i] is still the latent factor interaction term.

Using the above rules, both models follow a general format using gradient descent when undergoing training, with the only difference being if the bias terms are updated.

1. **Initialization:** In this step, we determine the number of unique users and items from the training set, and randomize P and Q to small, random values, and if bias is used, we initialize b_u b_i to zero and set b as the mean of observed values.
2. **Compute the Error:** The error quantifies the difference between the actual rating and the predicted rating, and it is calculated as $\text{rating}_{(u,i)}$ - $\text{prediction}_{(u, i)}$
3. **Updating Parameters:** Parameters will be updated using the rule below

$$P[u] \leftarrow P[u] + \alpha\Big(\text{error} \times Q[i] - \lambda P[u]\Big)$$

$$Q[i] \leftarrow Q[i] + \alpha\Big(\text{error} \times P[u] - \lambda Q[i]\Big)$$

   In addition to the above rule, if bias terms are used, we also include the following rules:

$$b_u[u] \leftarrow b_u[u] + \alpha\Big(\text{error} - \lambda b_u[u]\Big) \qquad b_i[i] \leftarrow b_i[i] + \alpha\Big(\text{error} - \lambda b_i[i]\Big)$$

$$b \leftarrow b + \alpha \times \text{error}$$

4. **Adaptive Learning Rate & Regularization:** Adaptive Learning Rate, increase alpha if RMSE increases, and decreases if not, allowing to better stabilize training if the model overshoots or gets stuck. Regulariation, lambda, included to prevent overfitting by penalizing large parameter values.

5. **RMSE Monitoring and Stopping Criterion:** After each epoch, compute the root mean square error. Continue iterating until the RMSE converges or a maximum number of iterations is reached.

To perform these steps, the following functions are defined.

- **fit(train):** Initializes parameters, runs gradient descent and monitors RMSE over iterations
- **ind_prediction(u, i):** Produces a single rating prediction for user u and item i
- **predict(X):** Takes a list of user-item pairs and returns predicted ratings for each pair.

Takeaways: The primary difficulty faced during implementation of both tactics are the fine tuning of parameters in order to find acceptable parameters to prevent over/underfitting to the data, as well as finding an appropriate adaptive learning rate to properly run gradient descent. Too large of a multiplier led to diverging results, and too small would prevent proper convergence.