SMAI ASSIGNMENT - 1

Observations from plots : I have drawn bar plots for categorical data and violin plots for continous data, the graphs suggest that the count of clear is very high in weather feature and windy count is less, the traffic level is mostly low medium, the number of orders are very less during night compared to other times of day. Mostly bike is preferred for delivering the food and the distribution of courier experience is mostly equal with slight variation. The distribution is mostly symmetrically spread around the mean of 10 km.

Analysis of lambda value on performance :

With increase in lambda values the loss is decreasing, but the decrease is for upto certain lambda value, after that because of underfit, the loss values are increasing. This is for both the regularization techniques.

Q2 : **Predicting Food Delivery Time**

Gradient Descent is a fundamental optimization technique used in machine learning and deep learning for minimizing loss function.

Differences between different Gradients

1. Batch Gradient Descent (BGD)

In Batch Gradient Descent, the entire dataset is used to compute the gradient of the cost function and update the parameters (weights) in each iteration.

- **Advantages**:
    - **Convergence to Global Minimum**: Since it uses the whole dataset, it tends to converge to the global minimum of the convex cost function.
    - **Accurate Gradient Calculation**: The gradient is computed accurately because it's based on the entire dataset.
- **Disadvantages**:
    - **Computationally Expensive**: For large datasets, it can be slow since it requires computing the gradient over the entire dataset in each iteration.
    - **Memory Intensive**: Requires storing the entire dataset, which can be problematic with large data.
    - **Slow Convergence**: Especially in cases where the data is large, each iteration can take time, making it slow.

2. Stochastic Gradient Descent (SGD)

In Stochastic Gradient Descent, instead of using the entire dataset, SGD updates the parameters using the gradient of the cost function for a single training example at a time.

- **Advantages**:

    - **Faster Convergence**: Since it processes one training example at a time, it updates the parameters more frequently and can converge faster.
    - **Memory Efficient**: Only a single data point is processed at a time, so it requires less memory.
    - **Escaping Local Minima**: The noisy updates can help the algorithm escape local minima and find a better global minimum.
- **Disadvantages**:
    - **Noisy Updates**: The updates are noisy because they're based on one training example at a time, which can cause the algorithm to oscillate and not converge smoothly.
    - **Convergence to Global Minimum is Uncertain**: Due to the randomness, SGD might not always converge to the global minimum, especially for non-convex functions.

3. Mini-batch Gradient Descent

Mini-batch Gradient Descent is a compromise between Batch Gradient Descent and Stochastic Gradient Descent. It divides the dataset into small batches (typically 32, 64, or 128 data points), and the gradient is computed for each batch, updating the parameters accordingly.

- **Advantages**:
    - **Efficient Computation**: It takes advantage of matrix operations (using libraries like NumPy or TensorFlow), making it faster than BGD while still not as noisy as SGD.
    - **Convergence**: It converges faster than BGD, and the updates are less noisy than SGD, helping in smoother convergence.
    - **Better Generalization**: Mini-batches provide some noise, helping avoid overfitting and improving generalization.
- **Disadvantages**:
    - **Tuning Required**: The batch size needs to be tuned for optimal performance. Too large a batch size can make it too close to BGD, while too small can make it behave more like SGD.
    - **Still Slower than SGD**: While it is faster than BGD, it's still not as fast as SGD in terms of processing time for individual updates.

Which One Converged Fastest?

- **Stochastic Gradient Descent converges the fastest initially** because it updates the weighta after processing each data point but it may fluctuate and not settle optimally.
- After stochastic gradient, Mini Batch Gradient is better because it is faster than Batch Gradient as it updates weights after every batch is processed.
- **Batch Gradient Descent (BGD) is the slowest** but useful for small datasets.

How did Lasso and Ridge regularization influence the model? What is the optimal lambda (amongst the ones you have chosen) for Lasso and Ridge based on test performance?

Lasso (L1) regularization encourages sparsity by shrinking some coefficients to zero, effectively selecting important features and reducing model complexity, making it useful for feature selection.

The unimportant features will get the weights to be zero with which we can determine them and remove them. Ridge (L2) regularization prevents overfitting by shrinking coefficients towards zero without making them exactly zero, leading to better generalization but retaining all features. While Lasso is ideal when many features are irrelevant, Ridge works well when all features contribute meaningfully. The optimal lambda for Lasso and Ridge are 0.5, 0.2 respectively where I got loss values less among all.

How does scaling of features affect model performance?

1. Gradient Descent (SGD, Batch, Mini-batch)

For **gradient descent-based algorithms**, scaling plays a crucial role in speeding up the convergence and ensuring efficient optimization:

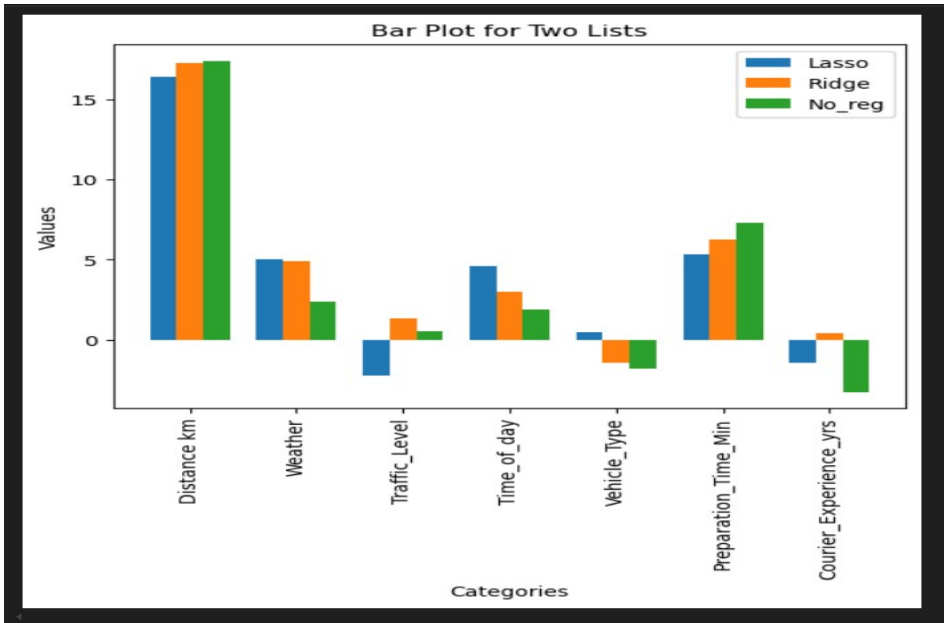
- **Without Scaling**: If features have different magnitudes (e.g., one feature ranging from 1 to 1000, and another from 0 to 1), gradient descent may converge slowly. This is because the updates to the model parameters are disproportionately influenced by the features with larger numerical ranges, which may cause the gradient descent to "zigzag" or take inefficient steps toward the optimal solution.
- **With Scaling**: Scaling the features (e.g., using standardization or normalization) ensures that all features are on a comparable scale. This results in more uniform gradient updates, which can significantly speed up convergence, especially with algorithms like **SGD**. Additionally, the learning rate can be optimized more effectively since it no longer needs to account for wildly different feature ranges.

2. Lasso and Ridge Regularization

Both **Lasso** and **Ridge regularization** are sensitive to the scale of the features:

- **Without Scaling**: When the features have different magnitudes, regularization (penalizing the coefficients) may unfairly prioritize the features with larger values. For instance, if one feature is much larger than others, its coefficient might shrink less due to the regularization term, even though it may not be more important for the model.
- **With Scaling**: Scaling ensures that all features are treated equally in the regularization process. **Lasso** and **Ridge** penalize all features in proportion to their magnitude, and scaling helps ensure that the penalty term is applied consistently across all features. This results in more effective regularization, preventing any one feature from dominating due to its larger scale, and can lead to better model performance and generalization.

Features Analysis :



Analysis of Feature Weights in Lasso, Ridge, and Non-Regularized Models

From the bar plot, we observe that the Lasso and Ridge models have different feature weight distributions compared to the non-regularized model.

- **Effect of Regularization:**
    - **Lasso (Blue):** Some feature weights are reduced significantly, and a few (e.g., "Vehicle_Type") appear close to zero. This suggests that these features are less important in predicting delivery time.
    - **Ridge (Orange):** Weights are reduced but still retain all features, unlike Lasso, which eliminates some. This indicates that Ridge focuses on minimizing overfitting without forcing weights to zero.
- **Feature Importance Analysis:**
    - **"Distance_km"** has the highest weight in all models, confirming that distance is the most crucial factor in delivery time.
    - **"Weather" and "Preparation_Time_Min"** also contribute significantly but are more controlled in Lasso and Ridge models.
    - **"Traffic_Level" and "Vehicle_Type"** have almost zero weights in Lasso, suggesting that they have little to no impact on delivery time.
    - **"Courier_Experience_yrs"** has a small negative or near-zero impact in all models, implying it is not a strong predictor.

Conclusion:

The best-performing model (likely Lasso) eliminated unimportant features ("Vehicle_Type"), leading to better generalization. Ridge retained all features but shrank weights, while the non-regularized model had larger weights, likely suffering from overfitting.

Q3 : KNN and ANN

When plotting histograms of bucket sizes for different numbers of hyperplanes, one problem that may arise is the **concentration of samples in a few buckets** when the number of hyperplanes is small. This happens because fewer hyperplanes lead to fewer distinct hash values, causing many data points to hash to the same bucket. As the number of hyperplanes increases, the buckets become more granular, and you may see a **wider spread of bucket sizes**, where the histogram becomes more evenly distributed but potentially with many **empty buckets** for higher numbers of hyperplanes.

Another issue is **imbalanced bucket distributions**, especially when using a small number of hyperplanes, which can result in a few buckets containing most of the data points while others have very few. For larger numbers of hyperplanes, the bucket sizes might become **too small** and the histogram might become less informative, as the data is spread across many more buckets with fewer samples per bucket. This can lead to a loss of insight from the histogram, making it harder to detect patterns in the data.

The metrics I got are

MRR: 0.3329733313110391

Precision@100: 0.1499859999999951

Hit Rate@100: 0.9964

How Metrics Change with the Number of Hyperplanes:

- **Increasing Hyperplanes**: More hyperplanes improve granularity, leading to better MRR value of relevant images. Increased hyperplanes reduce collisions, improving the distinction between relevant and irrelevant images, raising precision. More hyperplanes reduce collisions, increasing the likelihood of finding relevant images in the top 100 results.
- **Decreasing Hyperplanes**: Fewer hyperplanes result in more collisions, lowering the MRR value of relevant images. Less precise separation leads to more irrelevant images in the top 100, reducing precision. The hit rate may stay similar or decrease slightly as fewer hyperplanes cause more general hashing, affecting relevant image retrieval.

IVF :

The value of nprobe is 1

MRR: 0.9265684557243083

Precision@100: 0.8353769999999647

Hit Rate@100: 0.9985

The value of nprobe is 2

MRR: 0.9337073350074765

Precision@100: 0.8406339999999658

Hit Rate@100: 0.9991

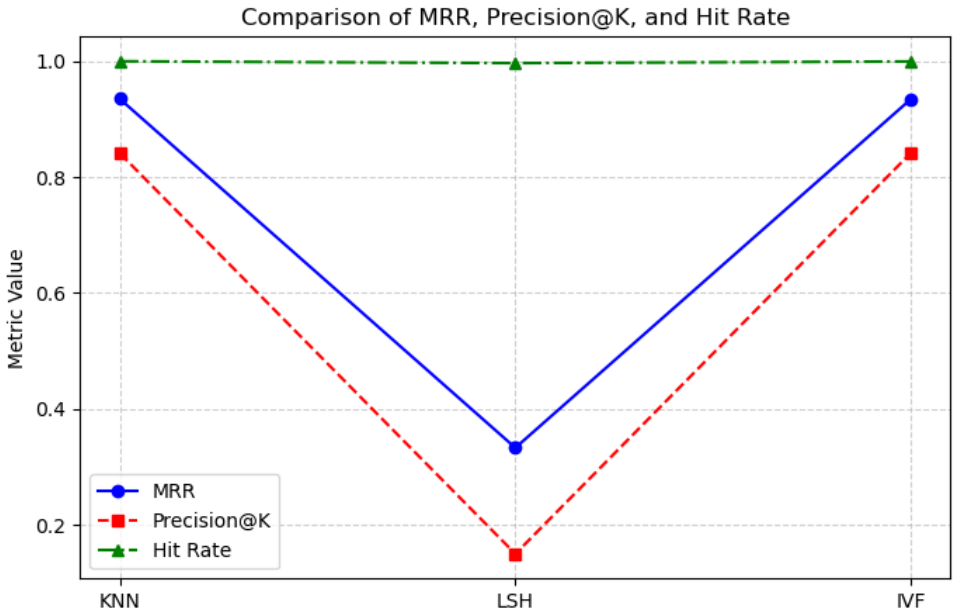
The value of nprobe is 3

MRR: 0.934433987656303

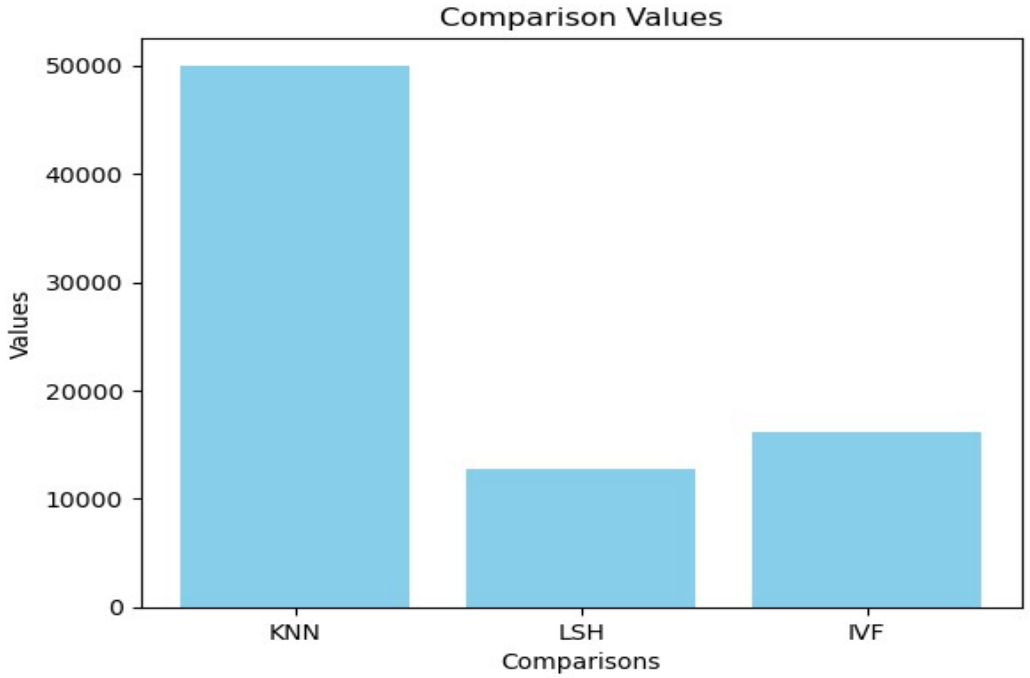
Precision@100: 0.8416579999999653

Hit Rate@100: 0.9993

Analysis :

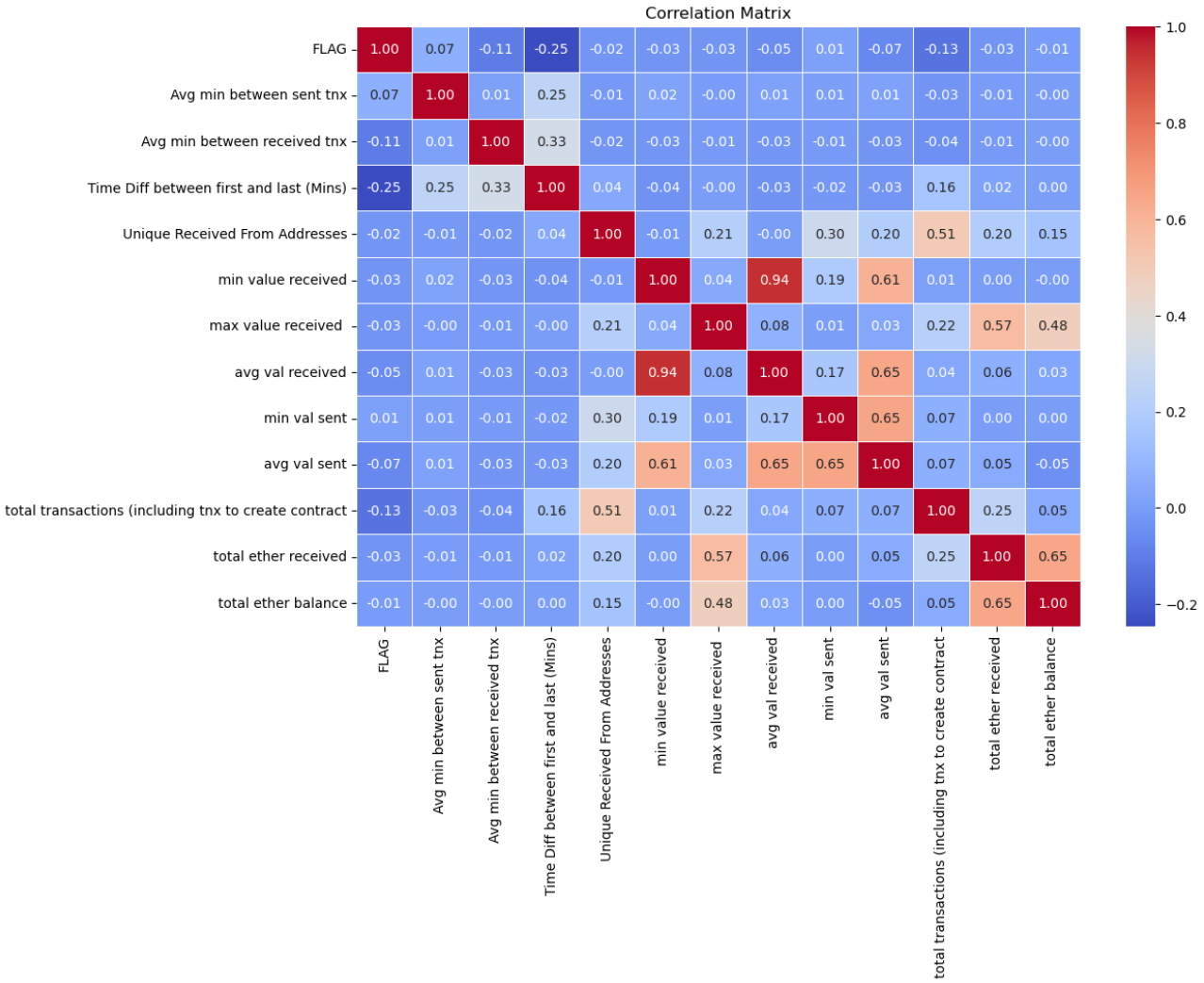


The common evaluation metrics across all three methods—Locality-Sensitive Hashing (LSH), K-Nearest Neighbors (KNN), and Inverted File (IVF)—are Mean Reciprocal Rank (MRR), Precision, and Hit Rate. The Hit Rate for all methods is nearly identical, indicating that at least one of the 100 nearest neighbors contains the true label for the test data point. However, Precision and MRR are significantly higher for KNN and IVF compared to LSH. This is because KNN directly calculates distances between the query point and all others, ensuring highly accurate nearest neighbors, while IVF efficiently narrows down the search space by focusing on relevant clusters. In contrast, LSH uses approximate matching via hash functions, leading to lower precision and MRR as it may group irrelevant points together, reducing the likelihood of retrieving true nearest neighbors.



The average number of comparisons is highest for K-Nearest Neighbors (KNN) because it computes the distance between the query point and every point in the dataset to identify the nearest neighbors. This exhaustive search leads to a large number of comparisons, especially in high-dimensional spaces. In contrast, Inverted File (IVF) reduces comparisons by partitioning the data into clusters and only searching within the most relevant clusters, thus significantly lowering the number of comparisons compared to KNN. Locality-Sensitive Hashing (LSH) performs the least number of comparisons, as it hashes data points into buckets and only compares points within the same or nearby buckets. However, LSH's reliance on approximate matching often leads to less accurate results, despite the reduced number of comparisons.

Crypto Detective :



Analysis of Correlation Matrix and Violin Plot

1. Correlation Matrix Analysis -

The correlation matrix helps us understand the relationships between different features. Below are the key observations:

- Strongly Correlated Features:

- total ether received and total ether balance are also highly correlated (~0.65), meaning they carry similar information.

- avg val received and min val received have a high correlation (>0.9) which might indicate dependency.

- Weakly Correlated Features with FLAG:

- total ether balance, and total ether received show very low correlation with FLAG (close to 0).

2. Violin Plot Analysis -

The violin plot helps visualize the distribution of features across different FLAG values.

Features with Clear Distinction:

- Time Diff between first and last (Mins), total transactions, and avg min between sent transactions show noticeable differences between FLAG = 0 and FLAG = 1. These features could be useful for classification.

- **Features with Overlapping Distributions**:

  - min value received, max value received, avg val received, total ether received, and total ether balance have highly overlapping distributions across both FLAG values. This suggests that these features might not significantly contribute to classification.

Conclusion: Features that Might Not Be Relevant for FLAG Classification

Based on the correlation and violin plot analysis, the following features might not be useful for classification:

- total ether balance

- total ether received

- min value received, max value received, avg val received

Removing or giving lower importance to these features could improve the model's efficiency and avoid redundancy.

Hence the below features are removed:

- total ether received

- avg val received

Comparison between scratch implementation and Scikit tree :

Scratch Implementation for Gini-Impurity:

The accuracy for train_Data is 0.9080472458328734

The accuracy for val_Data is 0.6066225165562914

The accuracy for test_Data is 0.5888378664782762

Scikit Tree Implementation :

The accuracy for train_Data is 0.9039629098134452

The accuracy for val_Data is 0.6163355408388521

The accuracy for test_Data is 0.6026139173436948

Analysis :

When comparing the custom decision tree implementation with scikit-learn's built-in version, the accuracy results are similar. The scratch implementation has 0.908 on the training set, while scikit-learn’s is 0.904. On the validation and test sets, scikit-learn performs slightly better (0.616 vs 0.607 on validation, and 0.603 vs 0.589 on test), likely due to additional features like pruning and optimized hyperparameters.

The computation time for the scratch implementation may be higher, as it lacks the optimizations present in scikit-learn, which is faster due to efficient implementation and parallel processing.

Sources of Variation :

Even with the same parameters, scikit-learn’s implementation is highly optimized in Cython, whereas the scratch implementation is likely slower due to the use of Python loops and less efficient algorithms for tasks like splitting, impurity calculation, and traversal.

While both models have the same maximum depth and minimum samples per split, scikit-learn’s implementation might include additional regularization mechanisms or more advanced splitting criteria, which can improve generalization, especially on unseen data.

Even though both models use Gini impurity for splitting, scikit-learn might employ more advanced techniques for selecting the best split more efficiently.

Scratch Implementation for Entropy:

The accuracy for train_Data is 0.8812230930566287

The accuracy for val_Data is 0.7496688741721854

The accuracy for test_Data is 0.6382903567643942

Scikit Tree Implementation :

The accuracy for train_Data is 0.8806711557567061

The accuracy for val_Data is 0.7421633554083885

The accuracy for test_Data is 0.6375838926174496

Analysis :

The custom decision tree implementation using Entropy and scikit-learn's built-in version show very similar results across the training, validation, and test sets, with accuracy differences being negligible. Both models perform similarly on the train set (around 0.881), while the scratch implementation slightly outperforms scikit-learn on the validation set (0.750 vs. 0.742). However, the test set accuracies are nearly identical (0.638 for both), indicating comparable generalization.

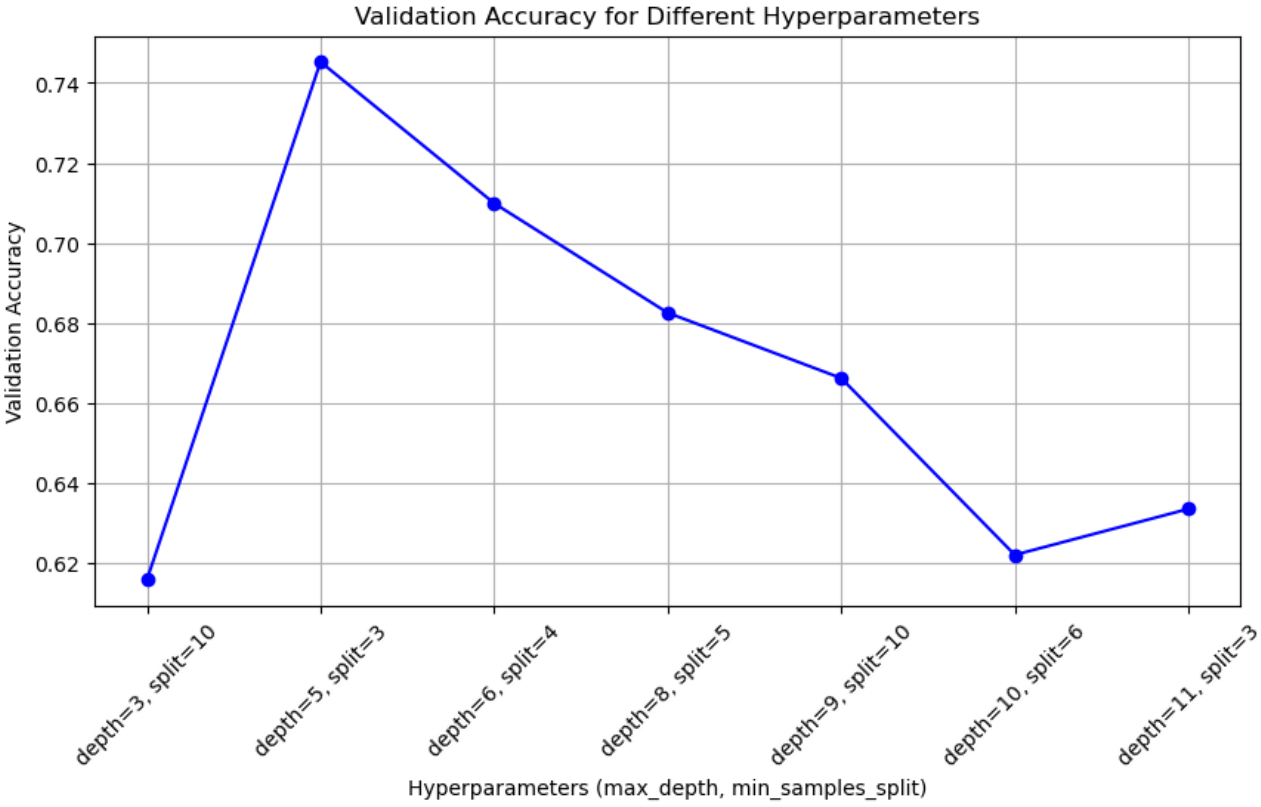
Despite the similar accuracy results, scikit-learn’s implementation benefits from optimization, making it faster and more efficient. It includes various performance improvements, such as better handling of edge cases and more efficient tree construction. The custom implementation is slower and may lack some of the advanced optimizations present in scikit-learn, resulting in longer computation times.

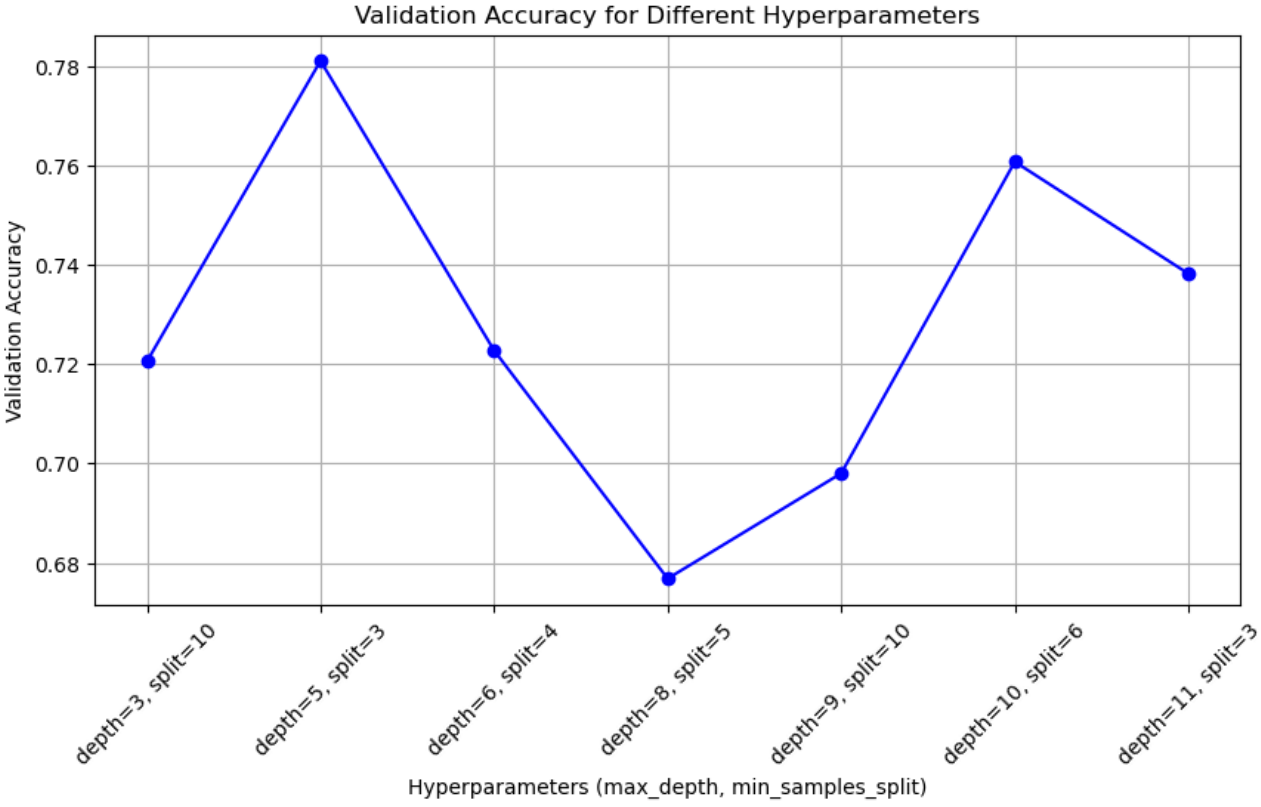
Key Decision tree Hyperparameters :

I have considered this set of hyperparameters to test their accuracies, these are the list of them

param_grid = [(3, 10), (5, 3), (6, 4), (8, 5), (9, 10), (10, 6), (11, 3)]

The below plots represent Accuracies of Gini-Impurity, Entropy


Validation Accuracy for Different Hyperparameters
Validation Accuracy
0.74
0.72
0.70
0.68
0.66
0.64
0.62
depth=3, split=10
depth=5, split=3
depth=6, split=4
depth=8, split=5
depth=9, split=10
depth=10, split=6
depth=11, split=3
Hyperparameters (max_depth, min_samples_split)


Validation Accuracy for Different Hyperparameters
Validation Accuracy
0.78
0.76
0.74
0.72
0.70
0.68
depth=3, split=10
depth=5, split=3
depth=6, split=4
depth=8, split=5
depth=9, split=10
depth=10, split=6
depth=11, split=3
Hyperparameters (max_depth, min_samples_split)

Possible Reasons :

The difference in validation accuracy between the Gini Impurity and Entropy criteria is likely due to how each method splits the data. Gini Impurity tends to favor splits that maximize class purity, leading to simpler trees that may generalize better in some cases. On the other hand, Entropy considers more detailed information gain, sometimes resulting in deeper trees that may fit the training data slightly better but risk overfitting.

Additionally, the trend in accuracy variation across hyperparameters suggests that certain depths and splits provide better generalization, while others lead to either underfitting (shallow trees) or overfitting (deep trees with low split thresholds). The differences in validation accuracy imply that Entropy may have led to higher accuracy in certain cases, possibly due to better handling of class distributions, but both criteria show the impact of hyperparameter tuning on model performance. Here just depth or just split cannot decide the accuracy of the tree data but both combinely does.

Q5 : **SLIC Implementation**

Hyperparameters – Clusters, Compactness

The **number of clusters (K)** directly affects the granularity of the segmentation. A lower K results in larger superpixels that retain more of the original image structure but may lose fine details. Increasing K generates smaller, more numerous superpixels, capturing finer textures and edges but potentially introducing over-segmentation. A balance must be maintained to avoid excessive fragmentation while preserving meaningful image regions.

Compactness (m) controls the trade-off between color similarity and spatial proximity when forming superpixels. A higher compactness value forces superpixels to be more spatially regular, leading to uniform, grid-like segments that may not align well with object boundaries. Conversely, a lower compactness value allows superpixels to adhere more closely to natural edges but may create irregular shapes, especially in textured regions.

Together, these parameters significantly impact the quality of segmentation. An optimal balance between K and m ensures that superpixels capture important structures while maintaining spatial coherence.

Using RGB Color Space :

When using the RGB color space to divide an image into small regions (superpixels), these regions don’t always match the actual shapes of objects. This happens because RGB doesn’t represent colors the way humans see them.

Colors in RGB can blend across object boundaries, causing uneven shapes, especially when brightness changes.

The Lab color space is better for this task because it separates brightness (L) from color (a, b), making edges clearer and segmentation more meaningful.

Lab space also ensures that differences in color appear more natural to human vision, leading to better-defined regions in an image.

Lab color space gives better and more accurate segmentation compared to RGB, because it aligns more closely with how we perceive color.

Optimizing the Video Segmentation :

I have used the concept of tolerance in Video Segmentation. Tolerance in video segmentation refers to the degree of flexibility allowed when distinguishing between different segments in a video based on changes in visual or motion features. It defines how much variation in color, texture, motion, or other attributes is acceptable before a new segment is created. So I will set a threshold value at first. After each iteration i will check the difference between h,w values and if the change goes below the threshold, I will break the loop there itself assuming that extra iterations cause very minimal changes that are negligible.
The average number of iterations used in the previous questions were 10

in this question the number of average iterations used were 7