2.Linear Regression

Test Set reports on each method:

	MSE	R2
Batch Gradient Descent	89.5771	0.8221
Stochastic Gradient Descent	88.5589	0.8242
Mini-Batch Gradient Descent	103.7572	0.7940

The three main types of **gradient descent algorithms** are:

1. **Batch Gradient Descent (BGD)**
2. **Stochastic Gradient Descent (SGD)**
3. **Mini-Batch Gradient Descent (MBGD)**

1. Batch Gradient Descent (BGD)

How it works:

- Uses the entire dataset to compute the gradient before updating the parameters.
- Updates the weights after computing the average gradient over all training examples.

Advantages:

More stable convergence (fewer fluctuations in the loss function).

Efficient use of vectorized operations (due to batch computation).

Disadvantages:

Requires large memory for big datasets (since all data is used at once).

Can be slow if the dataset is very large.

May get stuck in local minima (since updates are less noisy).

2. Stochastic Gradient Descent (SGD)

How it works:

- Updates the model parameters for each individual training example, rather than waiting for the entire dataset.

Advantages:

Faster updates, allowing quicker learning.

Can escape local minima due to the noise introduced by single-instance updates.
Works well for large datasets where full-batch updates are impractical.

Disadvantages:
High variance in updates, leading to oscillations in the loss function.
May never fully converge (keeps fluctuating around the optimal point).
Less efficient than batch methods due to lack of parallelization.

3. Mini-Batch Gradient Descent (MBGD)

How it works:

- Divides the dataset into small batches (e.g., 32, 64, or 128 samples per batch).
- Computes gradients and updates the parameters based on a batch instead of the whole dataset.

Advantages:
Balances efficiency and stability (reduces variance while maintaining speed).
Faster than full-batch gradient descent.
Can leverage hardware acceleration (like GPUs) efficiently.

Disadvantages:
Still requires tuning of batch size and learning rate.
May still get stuck in poor local minima (though less likely than BGD).

Which gradient descent method converged the fastest?

In my implementation of Linear regression with different methods the Stochastic gradient descent converged lower number of iterations than the other two methods. This is because in every iteration(epoch) I am updating the gradient at every sample of the input train set

When it comes to time the fastest to converge is Mini-Batch. It took less number of iterations than the Batch-gradient Descent and less time than the Stochastic gradient descent

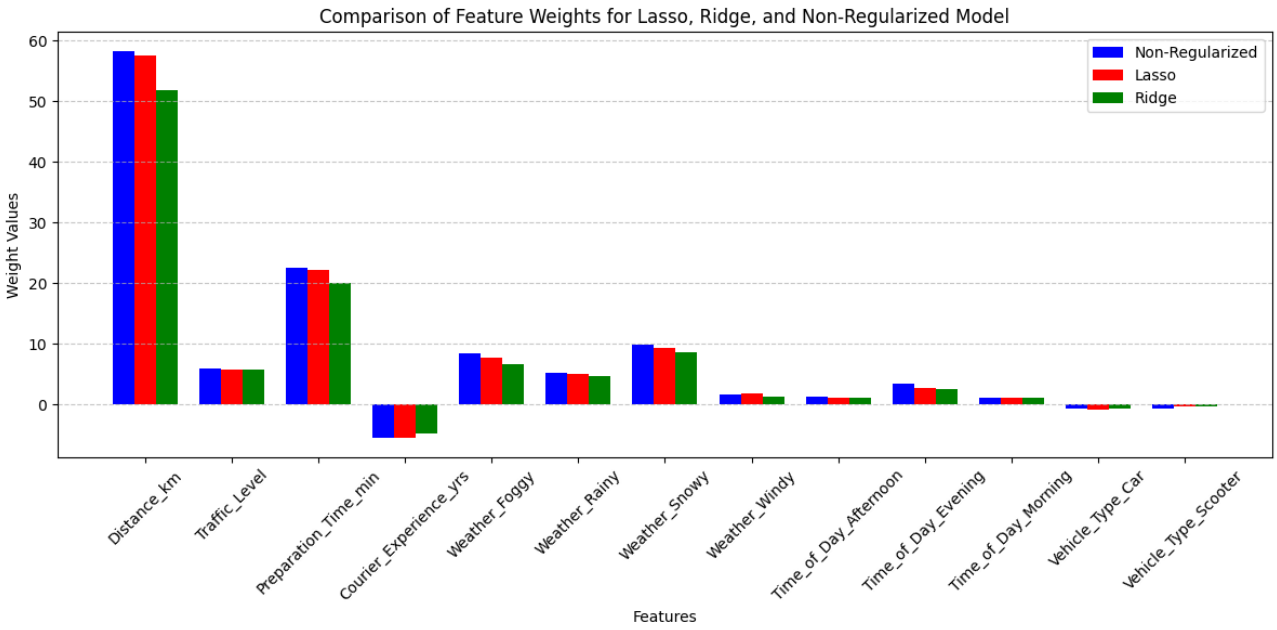
How did Lasso and Ridge regularization influence the model?

Lasso and Ridge did influence the model but only when their lambda value is very low close to 0.They adjusted the weights for the inputs where important feature given high value and the non important inputs given with low value.In this case almost every feature is important may that's the reason why lambda is near 0
Lambda for
Lasso : 0.0001
Ridge : 0.0001

How does scaling of features affect model performance?

The scaling didn't affect a lot this Is because the variance in the data is not much so the normalization or standardization didn't affect the model much
If the variance in the data is much then normalization and standardization will sure give better results that unscaled data

Using a Barplot, describe how the trained model weights (1 per feature) vary in case of best performing model for Lasso and Ridge, compared to non-regularized model (best).



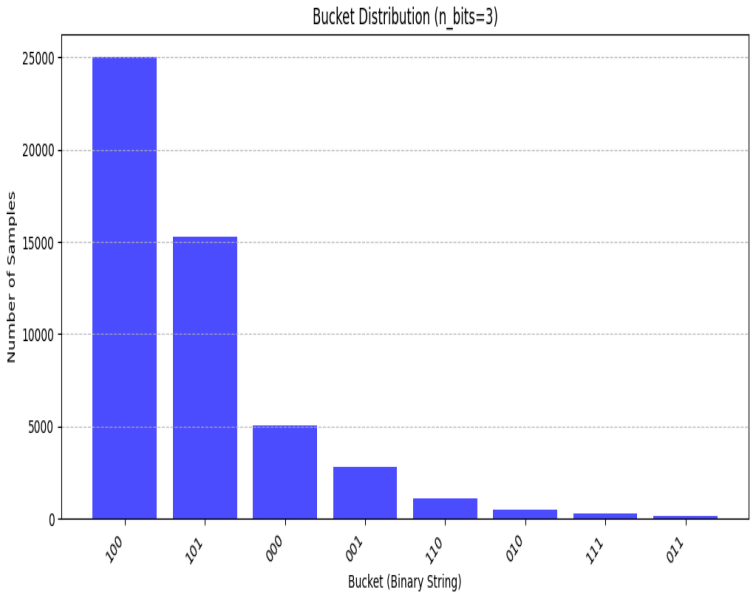
From the graph I can see that the for most of the features W_non > W_lasso > W_ridge

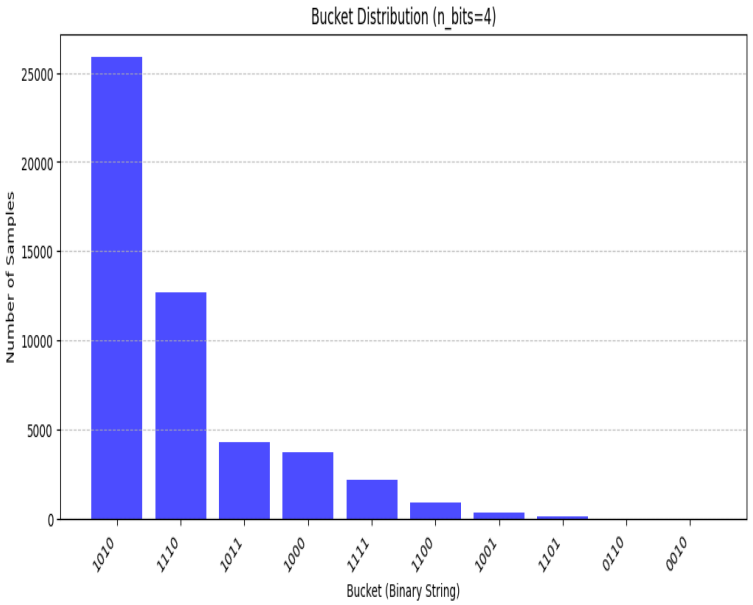
The Distance_km is the most important feature since its weight value is a lot when compared to others,Preparation time and weather also has some importance.

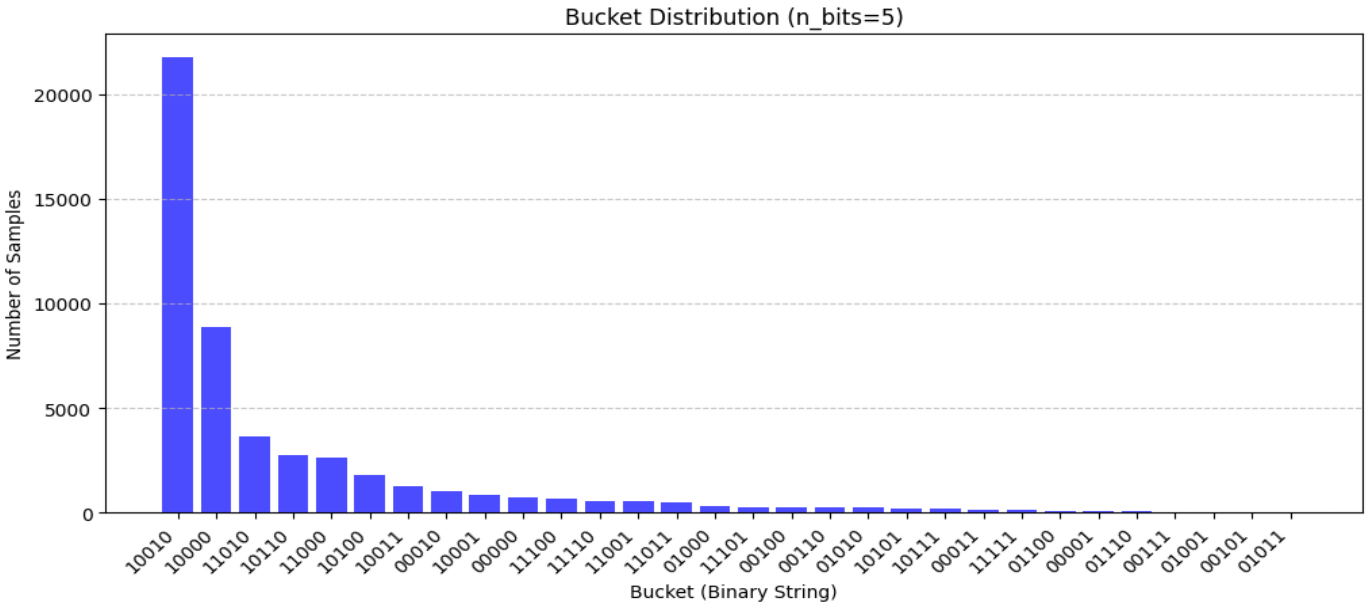
Time and Vehicle type weights are close to zero ,from this dataset the delivery time has less dependency on vehicle type and time.

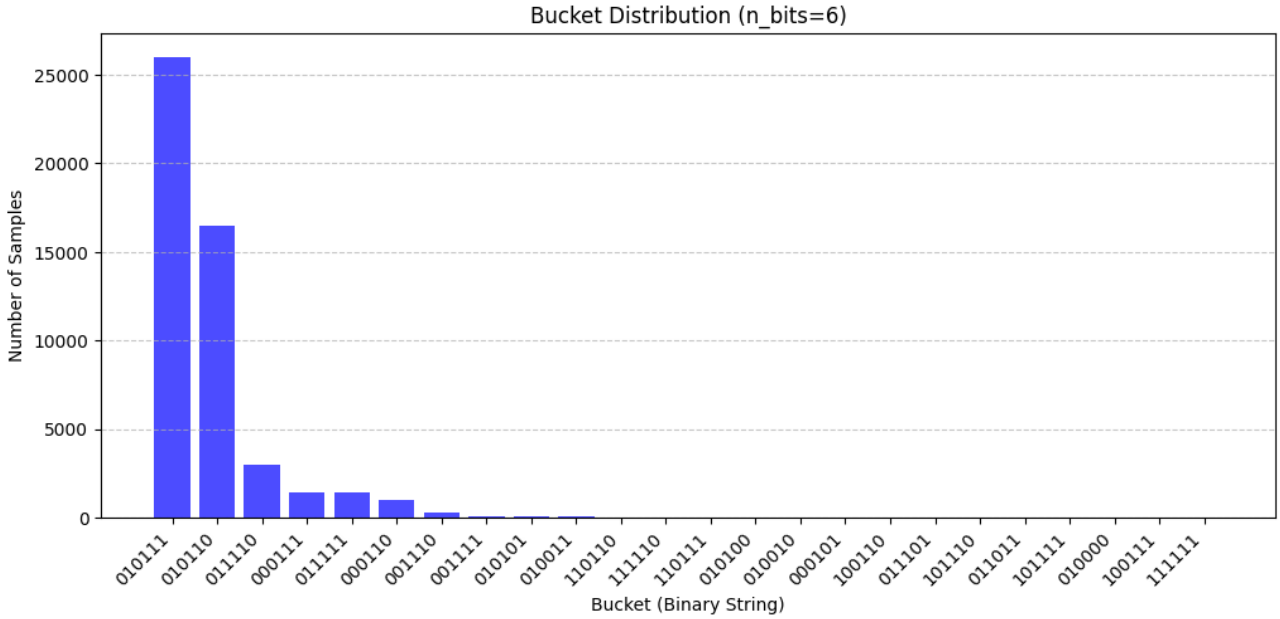
3.KNN

Bucket Distribution for different number of hyper planes:









Problems Noticed:

Uneven Distribution of the data points as most of the points are in one bucket and many buckets are almost empty.

High memory requirement as we have to of n bit we get 2 power n number of planes and each has to store the points belong to it.

Change of metrics based on the number of hyperplanes(number of bits):

	MRR	Precision	Hit Rate
4 bits	0.9273	0.8789	0.9523
5 bits	0.9252	0.8771	0.9511
6 bits	0.9192	0.8700	0.9448

1. Effect on MRR (Mean Reciprocal Rank):

Trend:

- **Increasing the number of hyperplanes** → MRR **increases** initially, then **saturates or decreases**.
- **Decreasing the number of hyperplanes** → MRR **decreases**.

Reason:

- MRR measures how quickly the correct nearest neighbor appears in the ranked list.
- More hyperplanes create **more selective (narrower) buckets**, so only very similar points end up in the same bucket.
- **Initially**, this improves ranking because it reduces false positives (irrelevant points).
- However, if too many hyperplanes are used, the dataset gets **overpartitioned**, meaning the true nearest neighbor might end up in a different bucket and never be retrieved, reducing MRR.

2. Effect on Precision:

Trend:

- **Increasing hyperplanes** → Precision **initially increases**, then **plateaus or drops**.
- **Decreasing hyperplanes** → Precision **decreases**.

Reason:

- Precision measures how many of the retrieved results are **actually relevant**.
- **With more hyperplanes,** LSH better separates dissimilar points, reducing false positives, increasing precision.
- **Too many hyperplanes** → Buckets become too small → Some relevant points get separated, lowering recall, which indirectly affects precision.
- **Too few hyperplanes** → Large buckets → Many irrelevant points retrieved → Precision drops.

3. Effect on Hit Rate:

Trend:

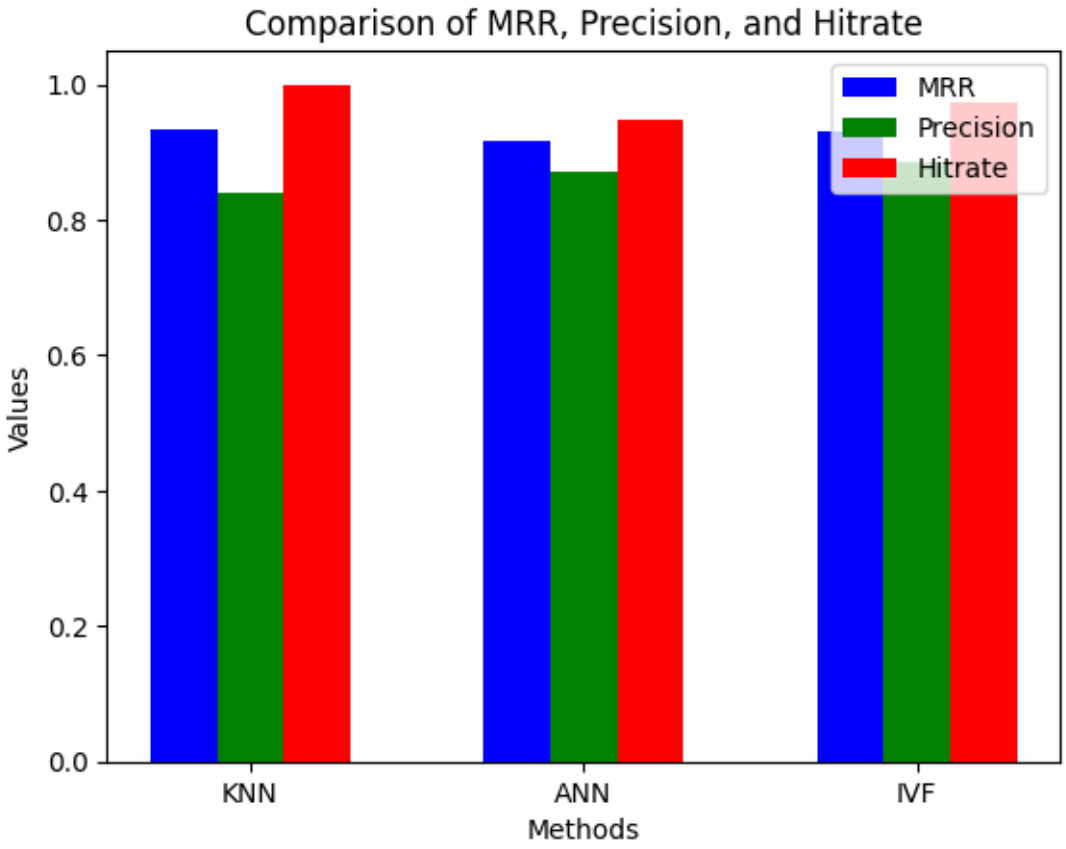
- **Increasing hyperplanes** → Hit Rate **decreases after a point**.
- **Decreasing hyperplanes** → Hit Rate **increases** but may include false positives.

Reason:

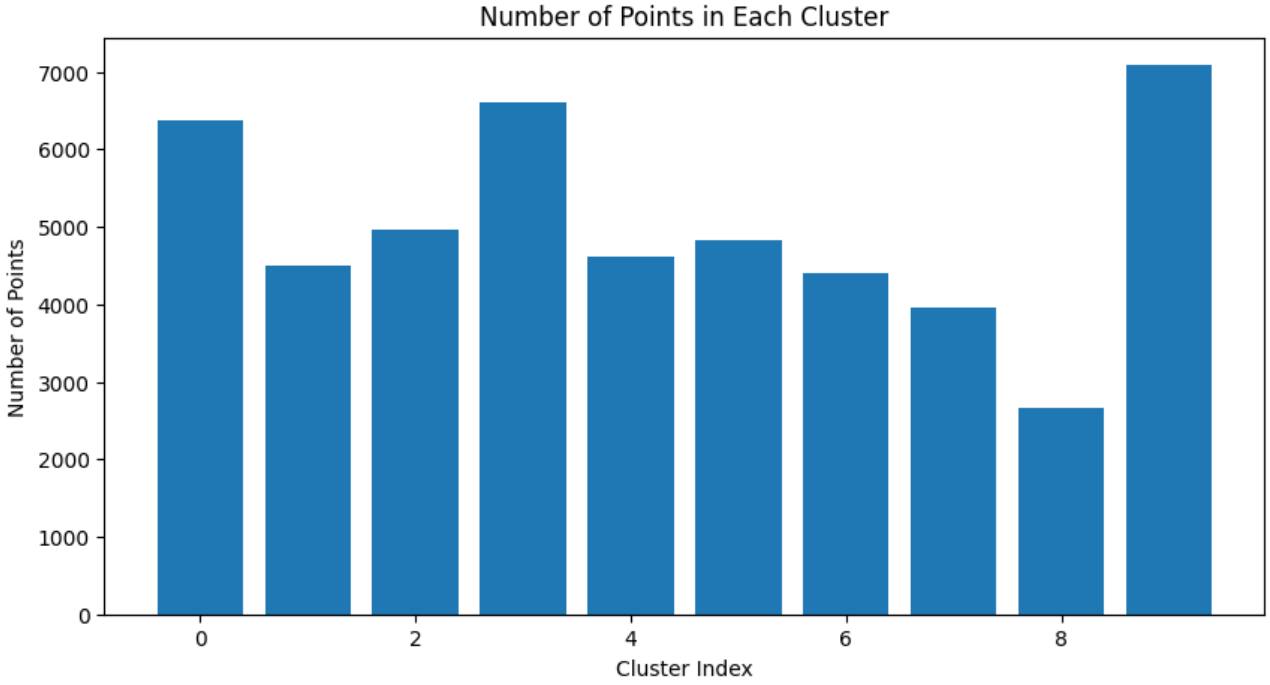
- Hit Rate measures whether at least one relevant result appears in the retrieved set.
- **Fewer hyperplanes** → Bigger buckets → Higher hit rate (more chances that at least one relevant item is retrieved).
- **More hyperplanes** → More refined separation → Some relevant points might be in another bucket, reducing hit rate.
- At an extreme, **very high hyperplanes** → **Each point is in its own bucket**, meaning no useful results are retrieved, and hit rate drops drastically.

IVF:

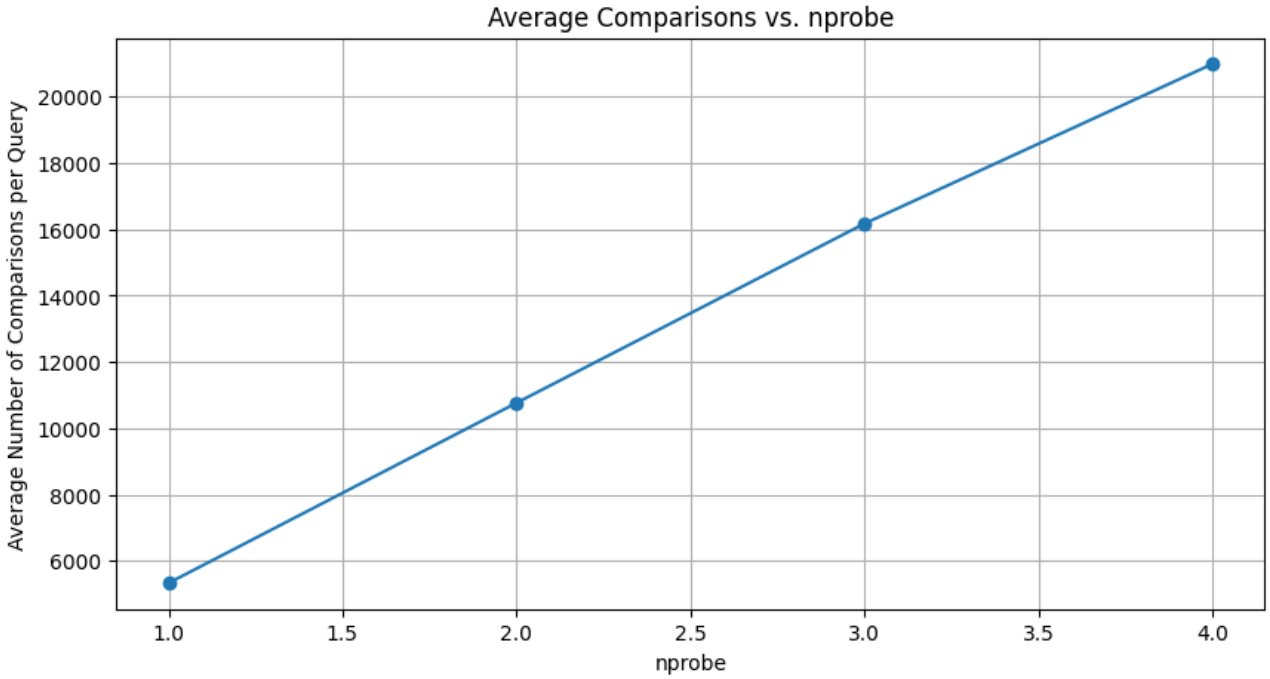
Comparing the three methods based on performance on the common task:



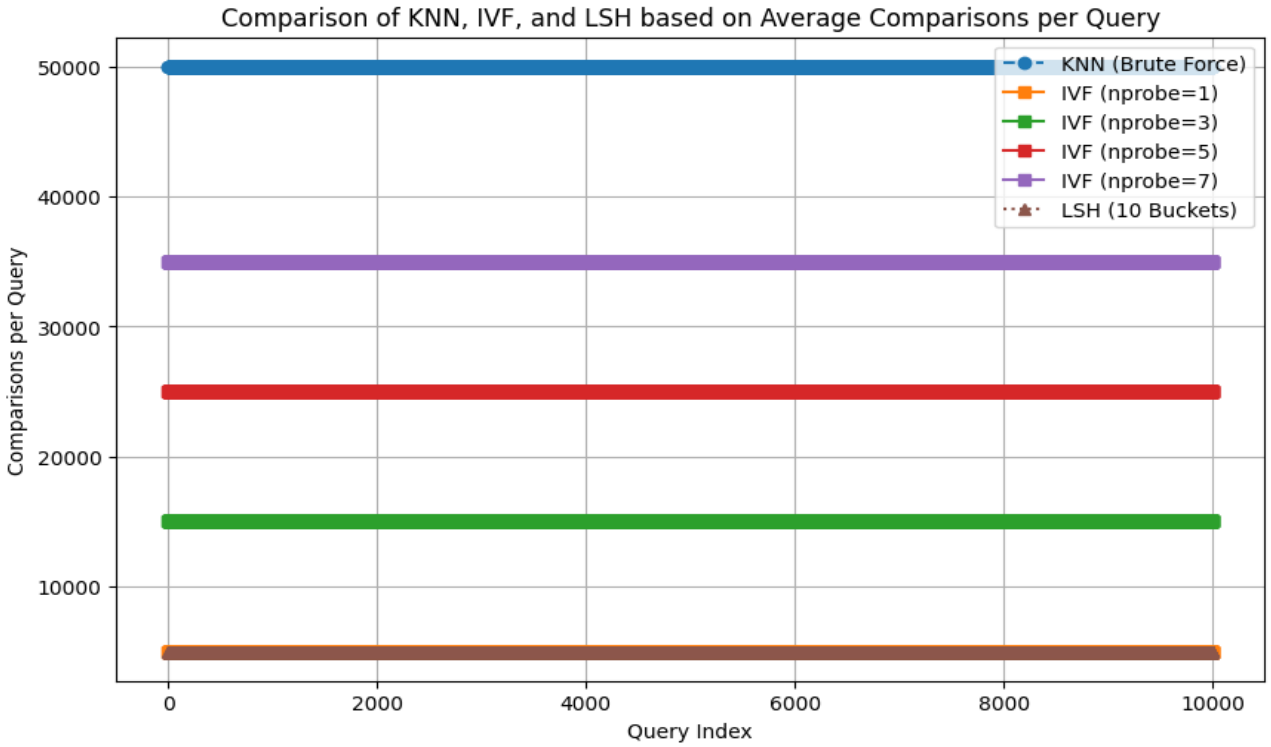
No.of poinst in each cluster:



average number of comparisons done for each query vs nprobe:

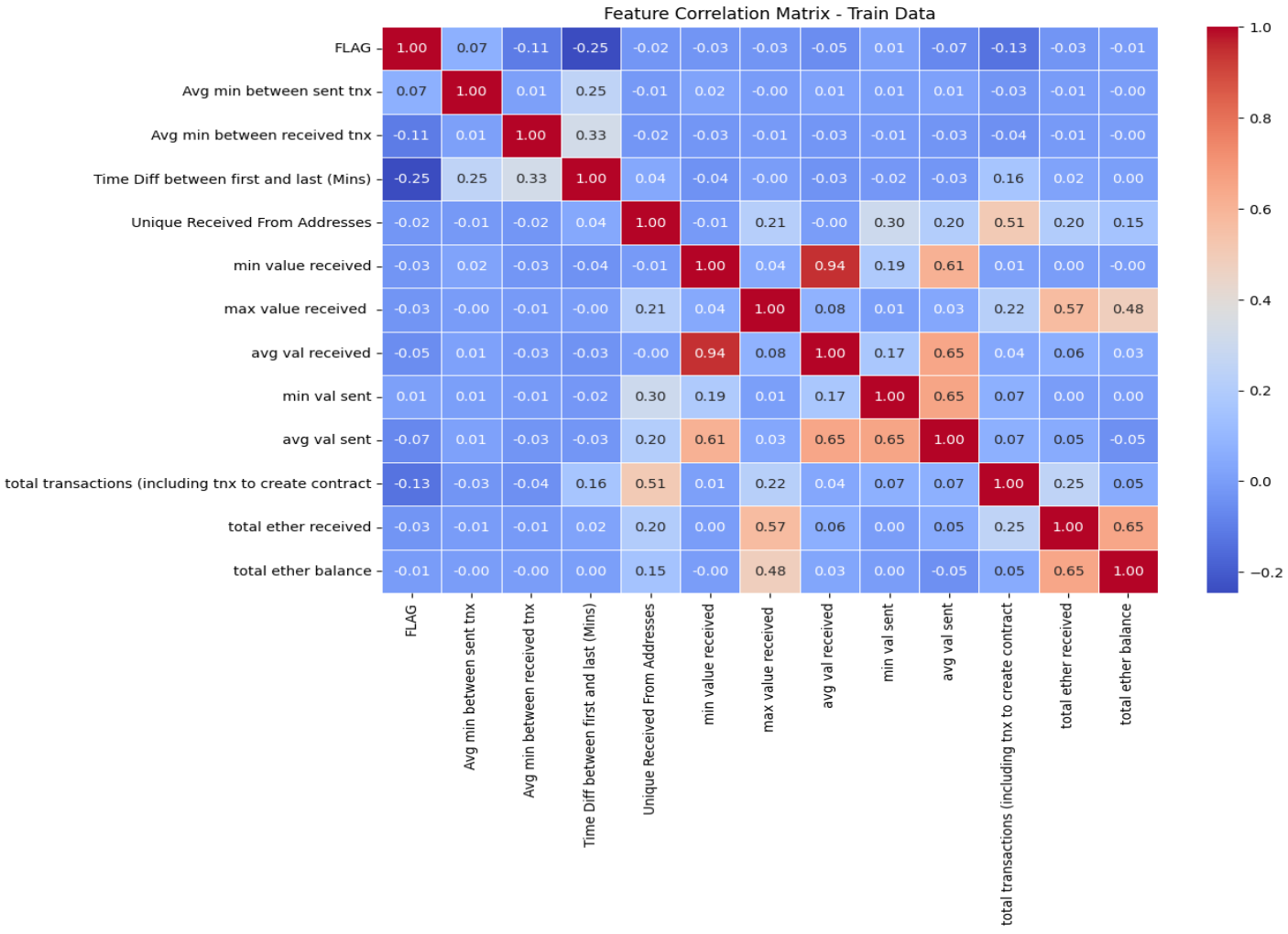


Comparing the three methods based on average number of comparisons done per query:

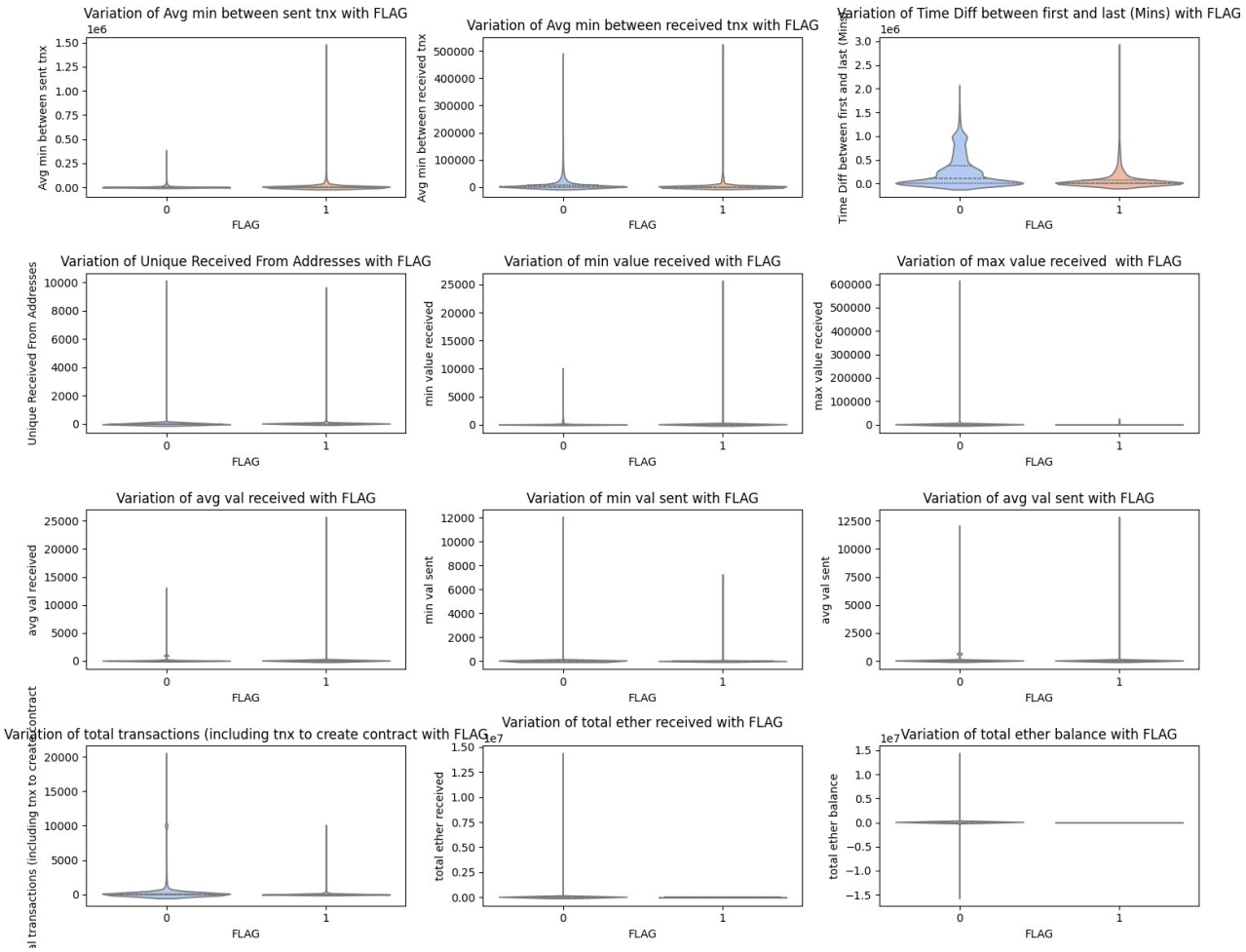


4 The Crypto Detective(Decision Trees):

Correlation matrix for the 13 features:



Plot the variation of other columns with respect to the FLAG column:



Comparing custom decision tree and scikit-learn's built-in implementation of decision tree:

	Train		validation		test	
	Entropy	Gini	Entropy	Gini	Entropy	Gini
Custom decision tree	0.8905	0.8905	0.9399	0.9399	0.9032	0.9032
Scikit learn	0.8883	0.8874	0.9399	0.9399	0.9032	0.8979

To calulate everything on trian validation and test for both entropy and Gini
my custom decision tree took 73.2658 seconds
Scikit learn library took 0.2237 seconds

My implementation of decision trees:

I will go through every feature and each unique sample of that feature and find the information gain and select the feature with a threshold value which has the least Information gain.This will be done at every node.

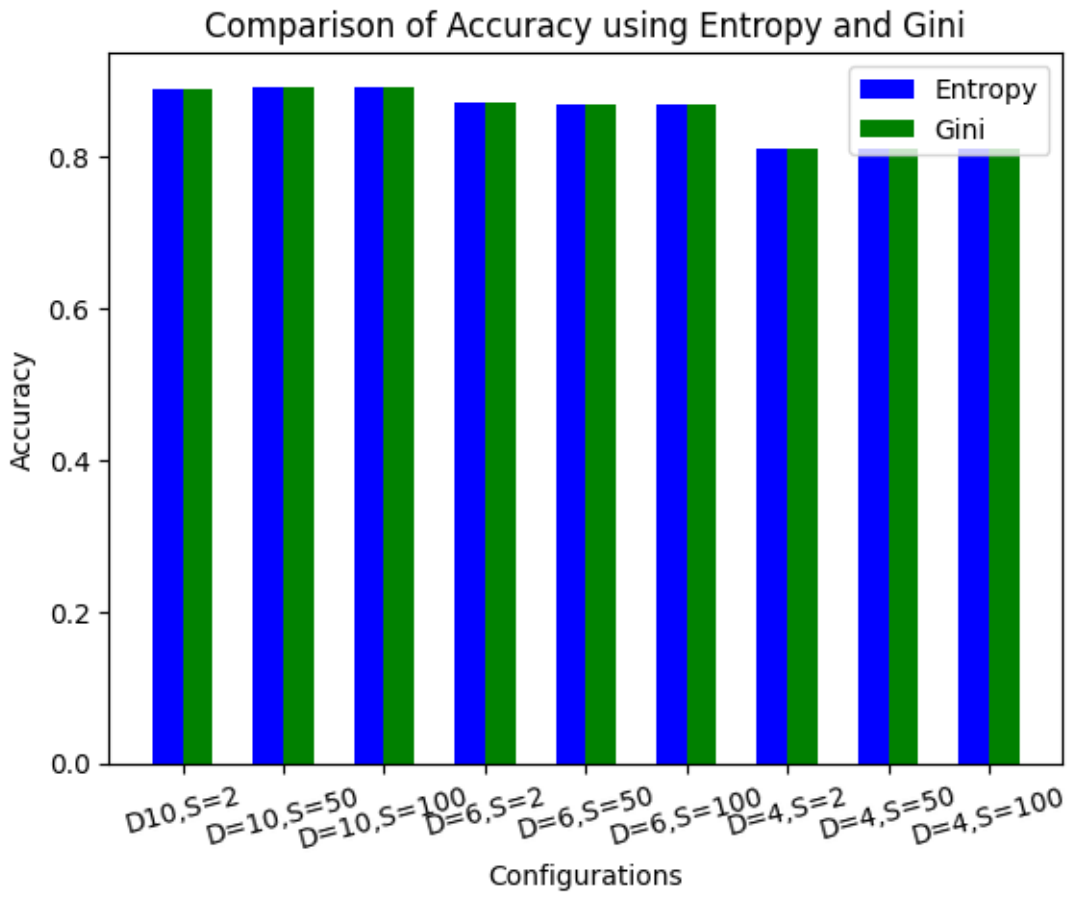
In case of hyperparameters I set the max_depth to 10 and the min_samples to split to 2, these are default settings

Scikit learn Implementaion on Decision trees:

- Uses an optimized version of CART (Classification And Regression Trees)
- For numerical features it sorts the values and finds optimal threshold
-

Here there is no max_depth the decision tree will keep on expanding until it has only one label in its leaf node and for the min_samples to split is set to 2 these are default settings

visualize performance across different model configurations on validation set:



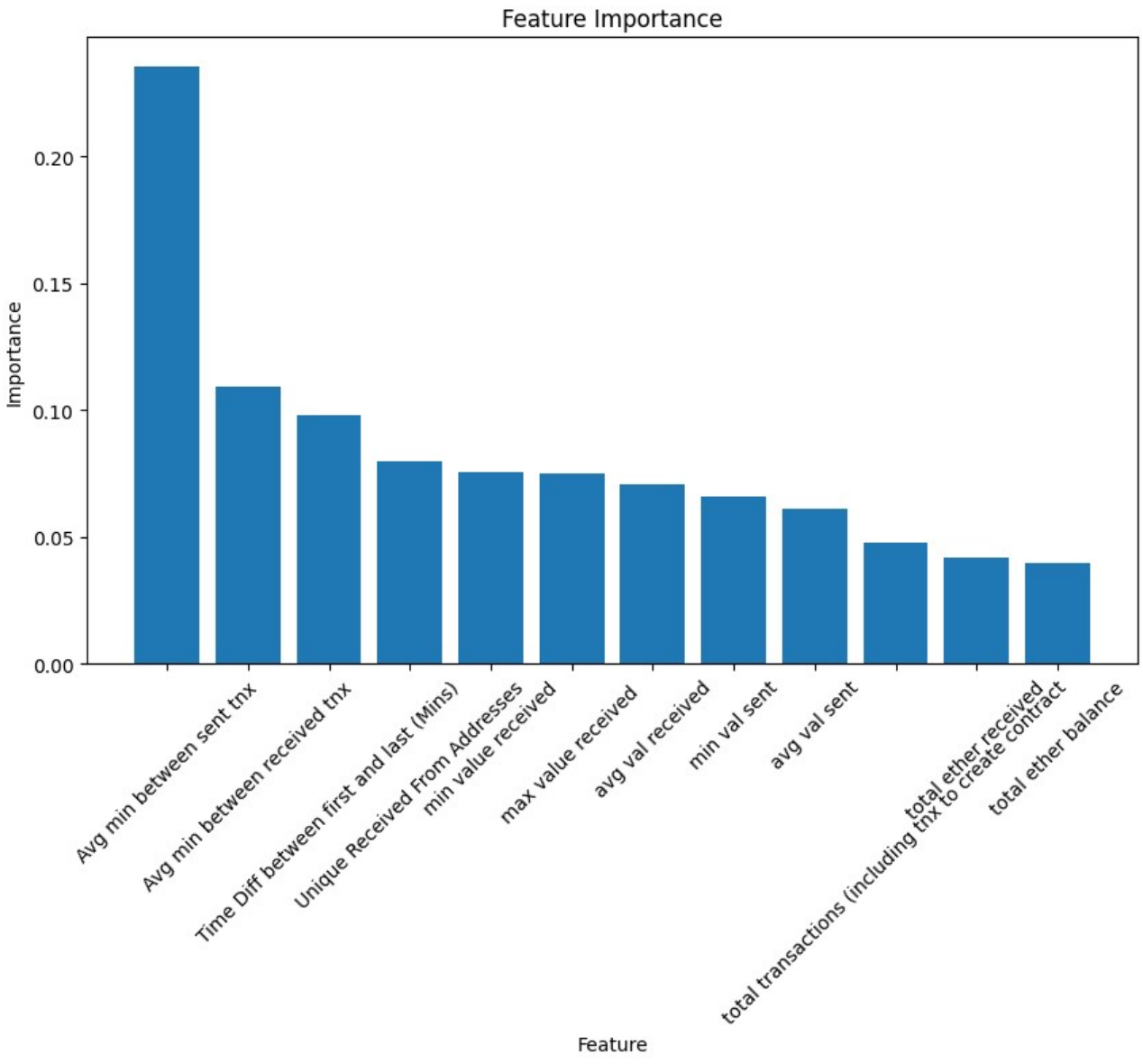
from the above graph I can see that as the depth increases the accuracy increases,
because;
When you allow the decision tree to grow **deeper**:

- The tree captures **more fine-grained patterns** in the data.
- It **reduces bias**, meaning it better fits the training data.
- The decision boundaries become **more complex**, leading to **higher accuracy on training data**.

and as the split increases the accuracy decreases reasons;
min_samples_split controls when a node is allowed to split.

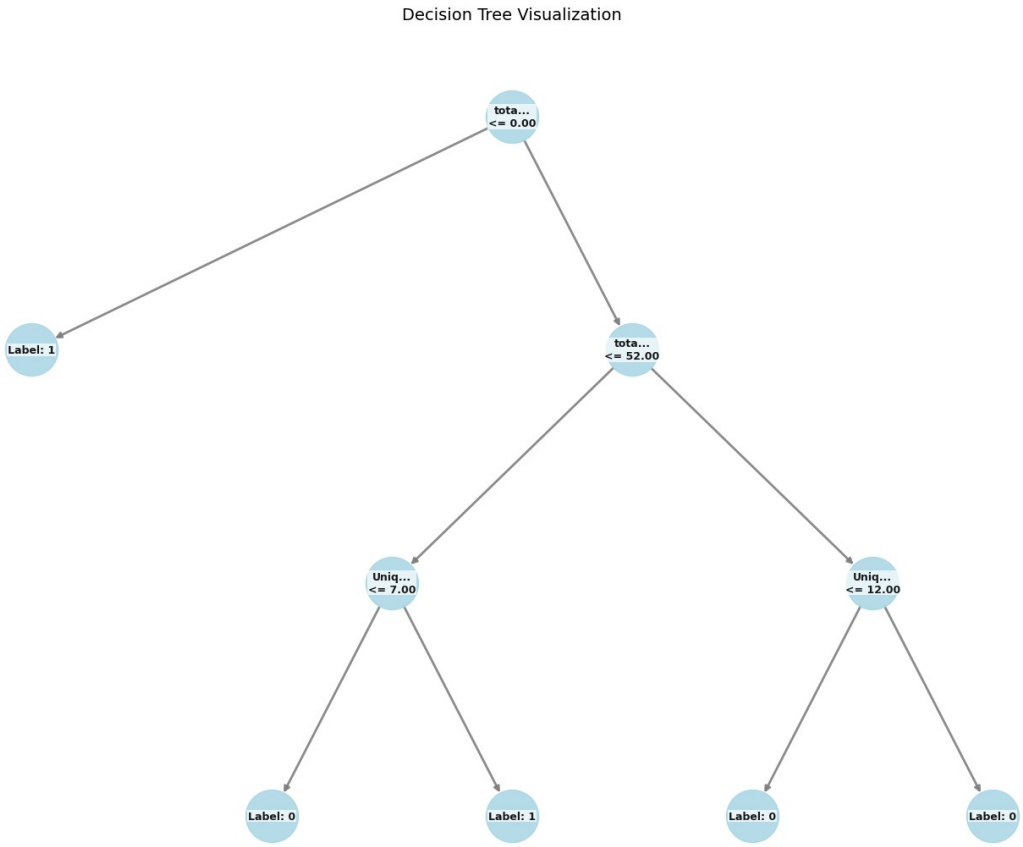
- Higher values (e.g., `min_samples_split=20`) mean a node must have at least 20 samples before splitting.
- This **prevents small splits**, resulting in a simpler tree.
- Simpler trees tend to underfit the training data, leading to lower accuracy.

Feature Importance Visualization:



Tree Visualization:

(max_depth = 3 and min_sample_split = 2)



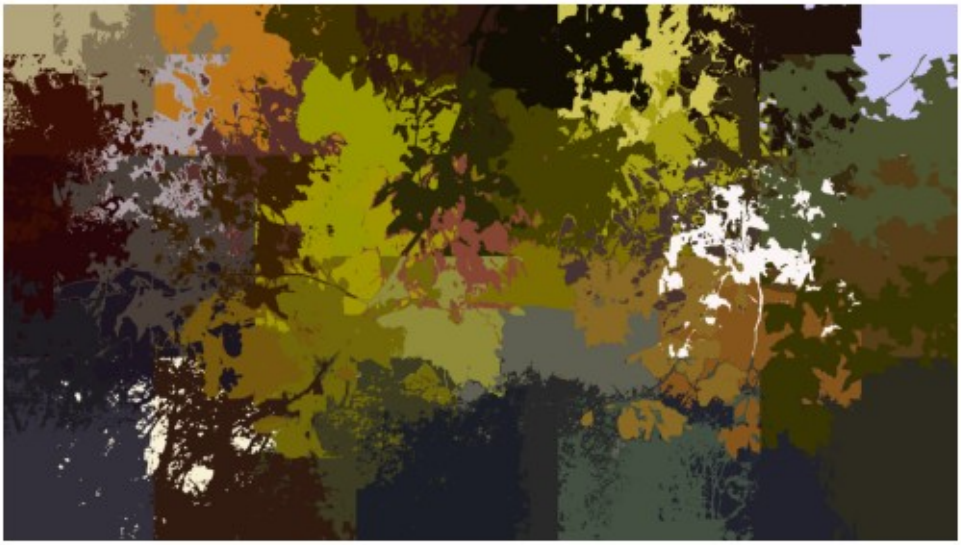
5.SLIC:

Changing the two hyperparameters(m and k):

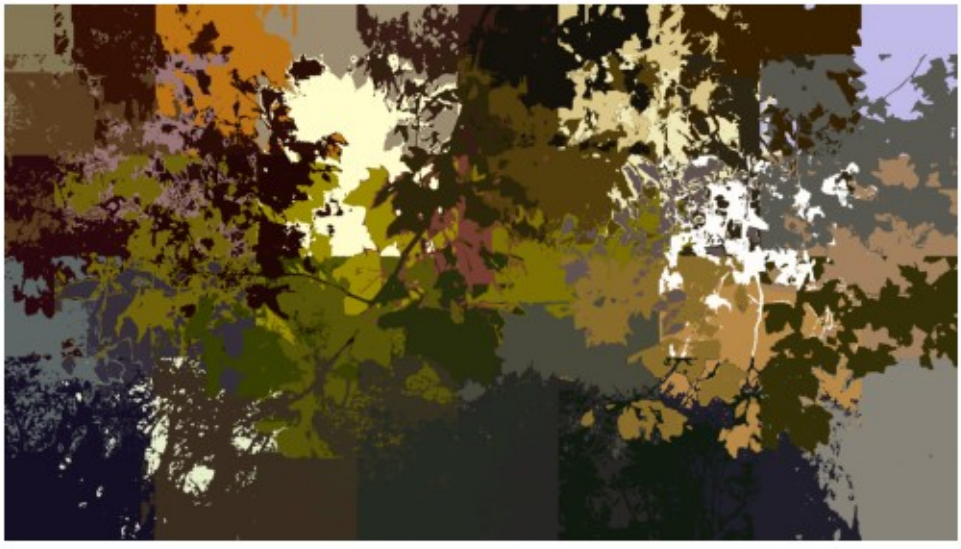
k = 100 and m = 20

k = 100 and m = 10

k = 50 and m = 20



k = 50 and m = 10



1. Effect of k (Number of Superpixels):

- determines how many superpixels the image will be divided into.

Effects:

Higher k (More Superpixels)

- Creates **smaller, finer segments**
- Preserves **more detail**
- **More computationally expensive**

Lower k (Fewer Superpixels)

- Creates **larger, coarser segments**
- Reduces details but speeds up computation
- **Might merge distinct regions**

2. Effect of m (Compactness Factor):

Definition:

- `m` controls the balance between **color similarity and spatial proximity**.
- Higher `m` → **More compact, regular** superpixels
- Lower `m` → **More irregularly shaped** superpixels

Effects:

✅ Higher m (More Compact Superpixels)

- Superpixels are **more regular and compact**
- Ignores some fine details in favor of **spatial coherence**
- Works better for **uniformly colored objects**

✅ Lower m (More Adaptive Superpixels)

- Superpixels **adapt better to edges and textures**
- More irregular shapes (preserve object boundaries better)
- Works well for **complex, high-texture images**

Iterations:

- After running **10 iterations**, the segmented image **barely changes** in subsequent iterations.
- This suggests that the algorithm **has mostly converged**, meaning further iterations bring diminishing returns.
- The **SLIC paper** also states that **10 iterations** are typically **sufficient** for convergence.
- The authors found that additional iterations **do not significantly improve segmentation**.
- **More than 10 iterations → Unnecessary computational cost** without noticeable improvements.

- **Less than 10 iterations** → Might result in **under-segmented** regions with **inconsistent superpixels**.

Without the Lab space just using the RGB:(k = 100 and m = 10)

from images we can see in the corner right the color is dark when used the lab space and light when used normal rgb this is because

SLIC (Simple Linear Iterative Clustering) is a superpixel segmentation algorithm that works best in a perceptually uniform color space. The choice of color space significantly impacts the segmentation quality.

Issues in RGB:

- **Not perceptually uniform**: The same Euclidean distance in RGB might not mean the same visual difference.
- **Sensitive to lighting**: Changes in brightness or shading affect segmentation quality.
- **Poor boundary adherence**: Objects with different brightness levels but similar colors might not be well-separated.

How SLIC Works in LAB:

- Instead of using raw RGB values, it converts the image to **LAB color space**, where:
  - **L** (Lightness): Represents brightness.
  - **A** (Green to Red spectrum).
  - **B** (Blue to Yellow spectrum).
- Distance computation in LAB is more accurate because the Euclidean distance in this space better represents **how humans perceive color differences**.
- LAB separates **color information** (A and B) from **brightness (L)**, making segmentation more robust.

Advantages of LAB Over RGB in SLIC:

- **Perceptual Uniformity** → More meaningful color clustering.
- **Better Superpixel Boundaries** → Adheres well to object edges.
- **Less Affected by Lighting** → Consistent segmentation in different lighting conditions.

Optimizing the video segmentation:

- The **first image** in each group undergoes **10 iterations** to ensure proper superpixel segmentation.
- The **remaining images in the same group** are processed with **4 iterations each**, leveraging the fact that they are visually similar to the first image.
- This strategy helps in **reducing computational cost** while maintaining **segmentation quality**.
- The number of iterations can be **adjusted dynamically** based on **group size, similarity threshold, or computational constraints**.

Comparing avg number of iterations used per frames:

1.Non Optimized approach:

There are a total of 11 images and every image uses 10 iterations
the final average number of iterations per frame is

(11* 10)/11 = **10**

2. Optimized approach:

Here for the for the 11 images I am grouping with 3 images per group and the first image is ran with 10 iterations and next 2 will have 4 iterations.

Up to 9 images there will be a total of 54 and 10^{th} image will have 10 iterations and at last 11^{th} image will have 4 iterations

so in total there will be 68 iterations.
The average number of iterations per frame = 68/11 = **6.1818**