Statistical Methods in AI (Spring 2025)

Assignment-1

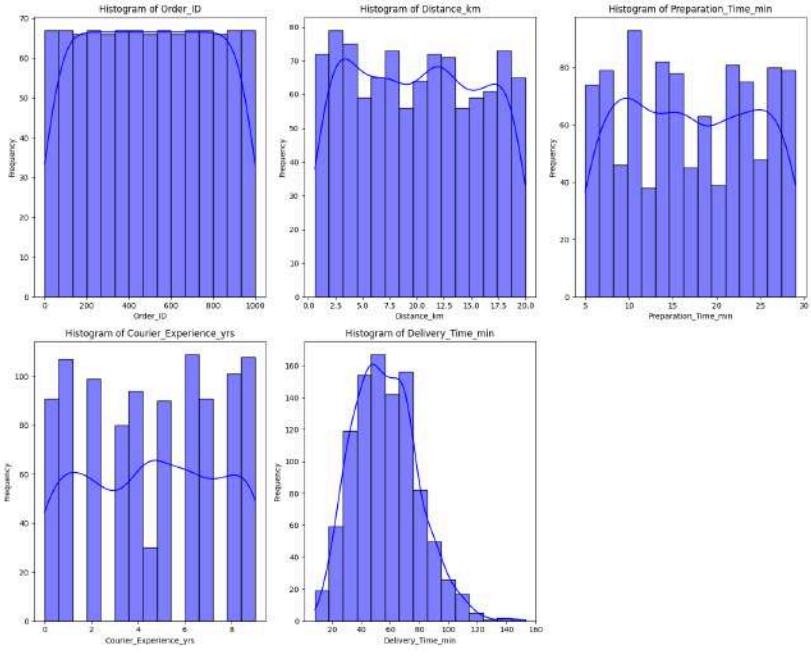
2022102068
Vaishnavi P

2 Predicting Food Delivery Time

2.1 Exploratory Data Analysis (EDA) :

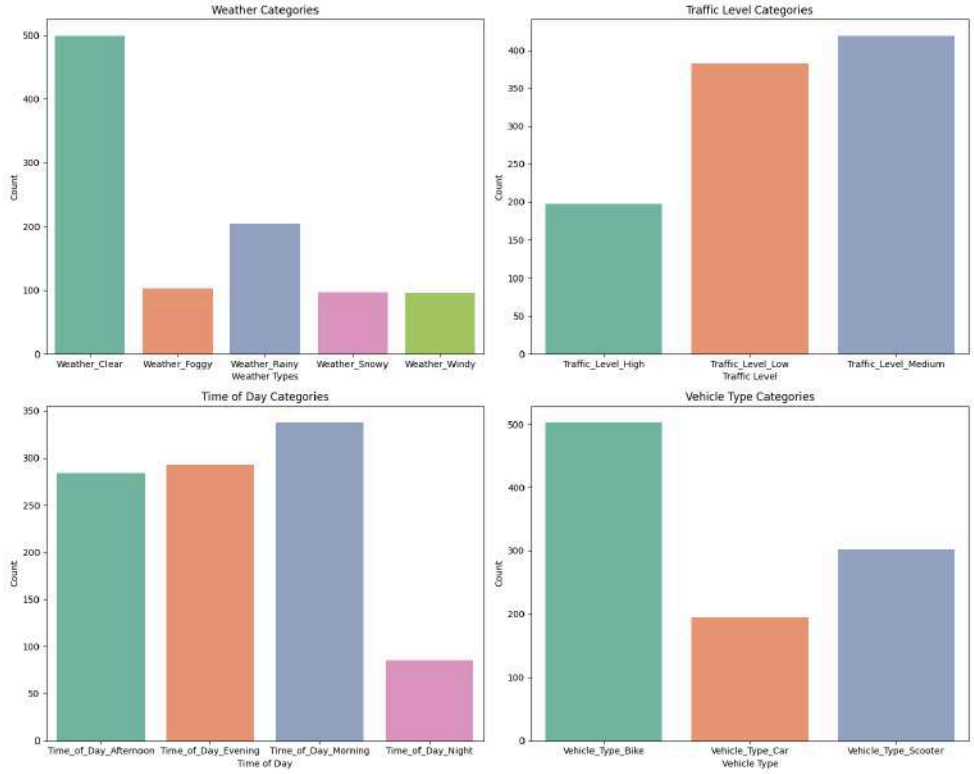
Considering the type of data each column has ,appropriate plots are considered.

a) **Numerical columns** - Distance_km, Preparation_Time_min, Courier_Experience_yrs, Delivery_Time_min
Histograms are plotted for numerical features because they group data into bins, allowing us to visualize the frequency distribution of data points within each range, which helps understand the distribution (e.g., normal, skewed, or uniform), identify central tendencies like mean and median, assess the spread or variance, and spot outliers or unusual patterns



b) **Categorical columns** - Weather, Traffic_Level, Time_of_Day, Vehicle_Type

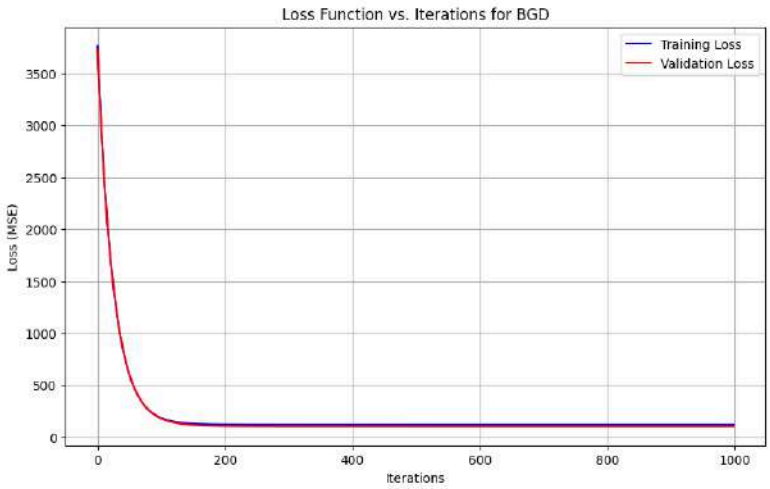
Bar plots are plotted for categorical features as they represent data in discrete categories by displaying the frequency or counts of each category, making them effective for visualizing comparisons between categories and identifying imbalances, such as dominant or rare categories.



Observations: All numerical columns can be seen to have almost uniform distributions and delivery_time can be seen to have less variance ,it can be inferred as within a short period of time most orders were delivered.
From categorical columns, the weather is clear most of the time during deliveries with medium to high traffic level. The orders were less at night compared to other times of day. Most of the delivery partners prefer bikes followed by scooters for delivery.

2.2 Linear Regression with Gradient Descent

a) Batch Gradient Descent: Iterations: 1000

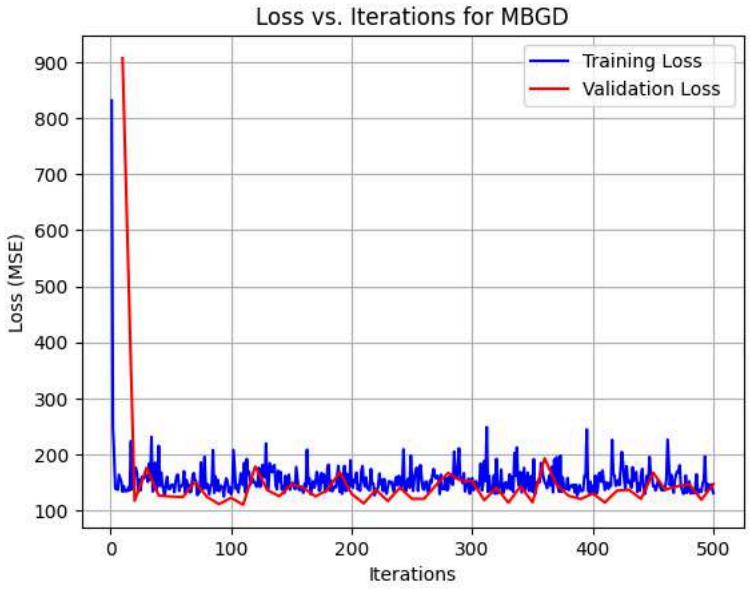



```
Iteration 10, Train Loss: 2649.1985, Validation Loss: 2639.3990
Iteration 20, Train Loss: 1807.1392, Validation Loss: 1809.7955
Iteration 30, Train Loss: 1246.1251, Validation Loss: 1253.1230
Iteration 40, Train Loss: 872.0801, Validation Loss: 879.2997
Iteration 50, Train Loss: 622.5431, Validation Loss: 628.0500
Iteration 60, Train Loss: 455.9865, Validation Loss: 459.0175
Iteration 70, Train Loss: 344.7700, Validation Loss: 345.1680
Iteration 80, Train Loss: 270.4801, Validation Loss: 268.3826
Iteration 90, Train Loss: 220.8412, Validation Loss: 216.5117
Iteration 100, Train Loss: 187.6648, Validation Loss: 181.4040
Iteration 110, Train Loss: 165.4860, Validation Loss: 157.5876
Iteration 120, Train Loss: 150.6560, Validation Loss: 141.3871
Iteration 130, Train Loss: 140.7379, Validation Loss: 130.3314
Iteration 140, Train Loss: 134.1037, Validation Loss: 122.7580
Iteration 150, Train Loss: 129.6653, Validation Loss: 117.5467
Iteration 160, Train Loss: 126.6955, Validation Loss: 113.9423
Iteration 170, Train Loss: 124.7080, Validation Loss: 111.4343
Iteration 180, Train Loss: 123.3777, Validation Loss: 109.6774
Iteration 190, Train Loss: 122.4872, Validation Loss: 108.4372
Iteration 200, Train Loss: 121.8910, Validation Loss: 107.5543
Iteration 210, Train Loss: 121.4917, Validation Loss: 106.9199
Iteration 220, Train Loss: 121.2243, Validation Loss: 106.4597
Iteration 230, Train Loss: 121.0452, Validation Loss: 106.1223
Iteration 240, Train Loss: 120.9252, Validation Loss: 105.8724
Iteration 250, Train Loss: 120.8448, Validation Loss: 105.6852
...
Iteration 970, Train Loss: 120.6814, Validation Loss: 105.0234
Iteration 980, Train Loss: 120.6814, Validation Loss: 105.0234
Iteration 990, Train Loss: 120.6814, Validation Loss: 105.0234
Iteration 1000, Train Loss: 120.6814, Validation Loss: 105.0234
```

Performance Metrics:

```
Test MSE: 62.751062, Test R²: 0.848866
```

b) Mini - Batch Gradient Descent: Iterations:500, batch_size = 15

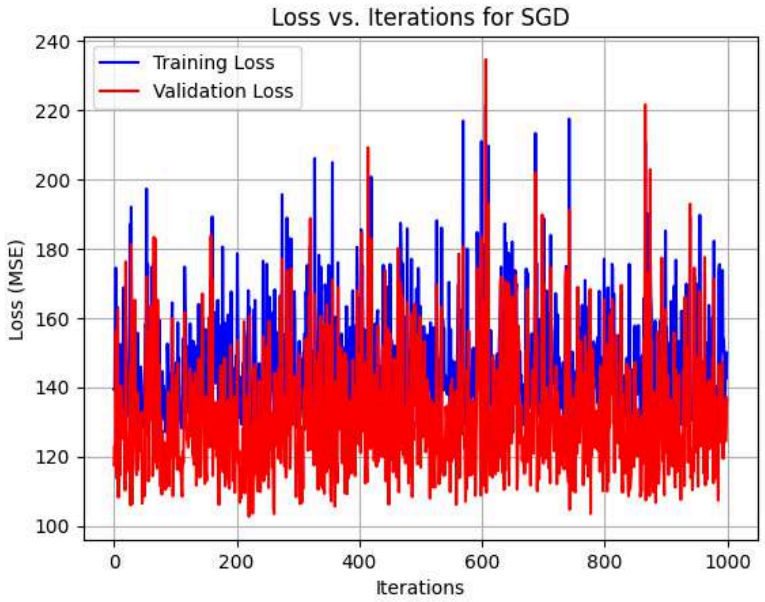



```
Iteration 1/500, Train Loss: 831.5146216657723, Validation Loss: 907.0264824061821
Iteration 11/500, Train Loss: 143.69759675536864, Validation Loss: 117.17014123697008
Iteration 21/500, Train Loss: 176.4717732586412, Validation Loss: 176.21461539798847
Iteration 31/500, Train Loss: 150.85491031822536, Validation Loss: 126.58797285809926
Iteration 41/500, Train Loss: 143.3568013692248, Validation Loss: 124.82968100620795
Iteration 51/500, Train Loss: 146.25971990858105, Validation Loss: 124.00421538544933
Iteration 61/500, Train Loss: 170.05808484645422, Validation Loss: 150.97069974945583
Iteration 71/500, Train Loss: 152.13319368372922, Validation Loss: 124.42269322446089
Iteration 81/500, Train Loss: 137.83868498213008, Validation Loss: 111.17132096391408
Iteration 91/500, Train Loss: 143.71108044505644, Validation Loss: 122.63583116331876
Iteration 101/500, Train Loss: 136.9110402018189, Validation Loss: 110.06643059000923
Iteration 111/500, Train Loss: 185.05639484201058, Validation Loss: 178.50385808076672
Iteration 121/500, Train Loss: 142.18838233286445, Validation Loss: 136.31599255803295
Iteration 131/500, Train Loss: 139.60461786509163, Validation Loss: 125.31234683537424
Iteration 141/500, Train Loss: 168.75588866230817, Validation Loss: 147.49812828295205
Iteration 151/500, Train Loss: 149.3999002416841, Validation Loss: 140.43355384625866
Iteration 161/500, Train Loss: 148.166096639789, Validation Loss: 124.90597379580954
Iteration 171/500, Train Loss: 168.80195491350523, Validation Loss: 136.70316519114027
Iteration 181/500, Train Loss: 181.78402939889634, Validation Loss: 168.41689015214553
Iteration 191/500, Train Loss: 165.3801067379147, Validation Loss: 129.55384861963407
Iteration 201/500, Train Loss: 140.51066818757536, Validation Loss: 112.55929992968667
Iteration 211/500, Train Loss: 142.7697724580421, Validation Loss: 137.0648978494357
Iteration 221/500, Train Loss: 134.50934143515852, Validation Loss: 116.79605587409667
Iteration 231/500, Train Loss: 152.43770939118417, Validation Loss: 140.60069827751047
Iteration 241/500, Train Loss: 138.10063059016917, Validation Loss: 120.90340471137519
...
Iteration 461/500, Train Loss: 155.5075131667653, Validation Loss: 143.21112602682877
Iteration 471/500, Train Loss: 168.32143893348228, Validation Loss: 146.4848636186419
Iteration 481/500, Train Loss: 130.48711685374164, Validation Loss: 119.17871046038398
Iteration 491/500, Train Loss: 148.62181898820094, Validation Loss: 147.18567880476525
```

Performance Metrics

```
Test MSE: 66.25576836393387, Test R²: 0.8404250649228378
```

c) Stochastic Gradient Descent : Iterations=20




```
Iteration 0: Train Loss: 139.3779, Validation Loss: 117.5584
Iteration 10: Train Loss: 135.4541, Validation Loss: 114.6388
Iteration 20: Train Loss: 144.4640, Validation Loss: 115.4900
Iteration 30: Train Loss: 138.0067, Validation Loss: 106.4426
Iteration 40: Train Loss: 131.2820, Validation Loss: 122.3888
Iteration 50: Train Loss: 137.6188, Validation Loss: 112.6948
Iteration 60: Train Loss: 154.0028, Validation Loss: 163.5459
Iteration 70: Train Loss: 133.0112, Validation Loss: 116.6108
Iteration 80: Train Loss: 132.9397, Validation Loss: 117.8428
Iteration 90: Train Loss: 127.8751, Validation Loss: 111.5259
Iteration 100: Train Loss: 141.1941, Validation Loss: 136.4044
Iteration 110: Train Loss: 128.1931, Validation Loss: 111.1976
Iteration 120: Train Loss: 151.6091, Validation Loss: 123.8254
Iteration 130: Train Loss: 133.1232, Validation Loss: 117.7138
Iteration 140: Train Loss: 130.0119, Validation Loss: 113.9916
Iteration 150: Train Loss: 137.1666, Validation Loss: 115.9713
Iteration 160: Train Loss: 189.2140, Validation Loss: 171.3084
Iteration 170: Train Loss: 129.5402, Validation Loss: 109.4784
Iteration 180: Train Loss: 158.6448, Validation Loss: 148.4803
Iteration 190: Train Loss: 157.8045, Validation Loss: 152.2219
Iteration 200: Train Loss: 144.0007, Validation Loss: 134.9898
Iteration 210: Train Loss: 146.8590, Validation Loss: 127.9578
Iteration 220: Train Loss: 137.1386, Validation Loss: 102.6419
Iteration 230: Train Loss: 165.1171, Validation Loss: 146.4381
Iteration 240: Train Loss: 142.1688, Validation Loss: 130.5689
...
Iteration 960: Train Loss: 133.8727, Validation Loss: 127.9519
Iteration 970: Train Loss: 163.6579, Validation Loss: 142.0543
Iteration 980: Train Loss: 132.0392, Validation Loss: 113.7305
Iteration 990: Train Loss: 153.9830, Validation Loss: 133.5941
```

Performance Metrics:

```
Test MSE: 86.70457762984714, Test R²: 0.7911747506997875
```

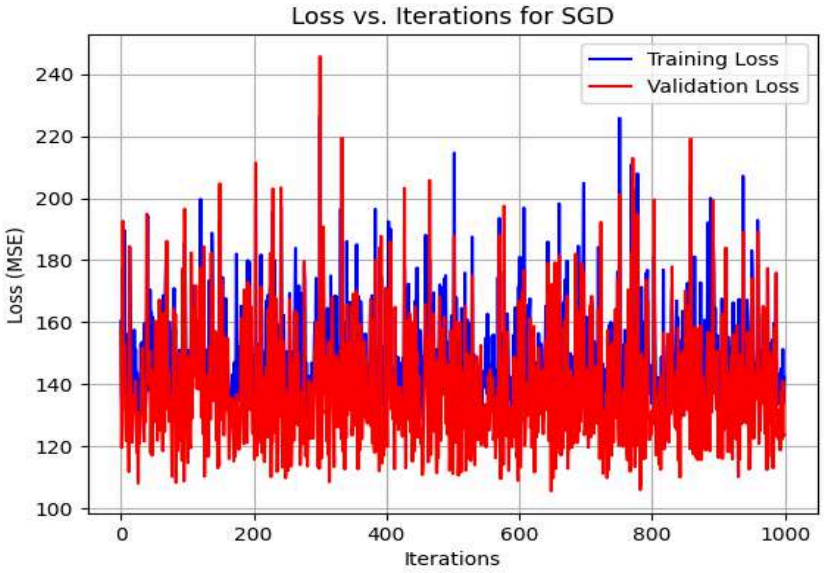
Observations:

Mini-batch GD performs better than stochastic GD and Batch GD. It can be seen that for around 1000 iterations Batch GD almost reached minimum MSE whereas Mini-batch achieved this in 500 iterations with batch size of 15. Stochastic achieved a minimum MSE for 1000 iterations.

So the loss function reduces drastically for Mini-batch GD with few epochs making it the best method for the given dataset.

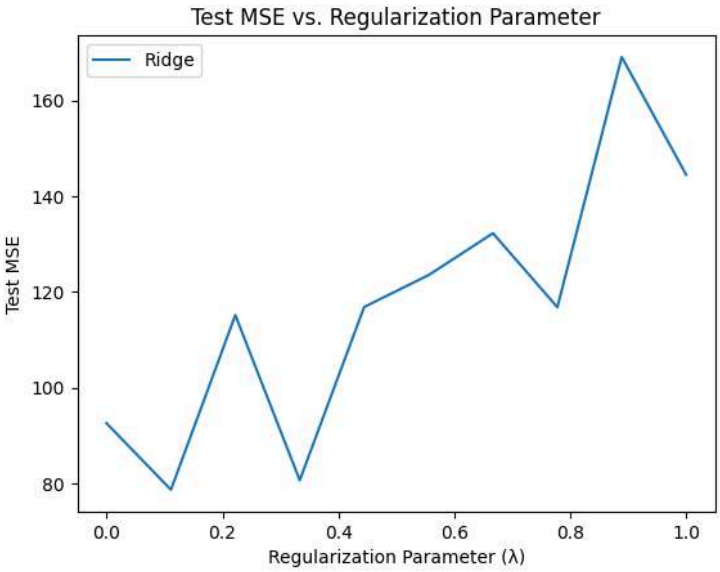
2.3 Regularization:

a) Ridge Regression

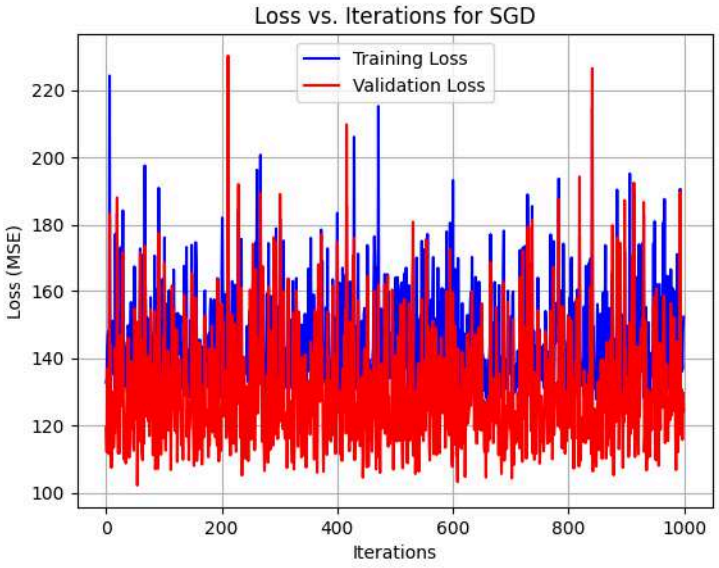


Performance Metrics:

```
Ridge Test MSE : 82.9599482460739, R²:0.8001935728425086
```

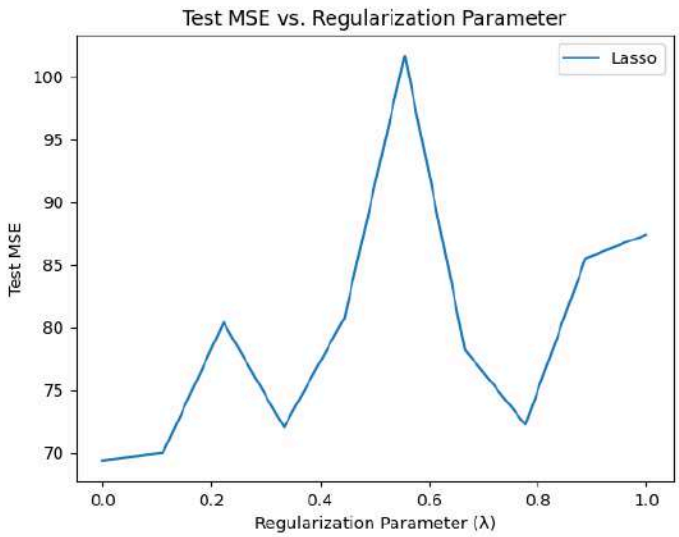


b) Lasso Regression



Performance Metrics:

```
Lasso Test MSE : 77.67612390882776, R²:0.812919497639634
```



Observations:
For the same value of lambda lasso worked better than ridge regression, one of the reasons can be that lasso can reduce weights to zero depending on the importance of features.

Using Lasso or Ridge regression with SGD, the test MSE vs. lambda plot can show unevenness because of the randomness introduced by using random samples. This randomness leads to noisy updates in the model parameters, causing fluctuations in the test MSE values.

Additionally, the learning rate in stochastic methods greatly affects convergence. If it's too high, the model overshoots optimal solutions, leading to instability. If it's too low, the model might not converge well, causing inconsistent performance for different lambda values.

In Lasso, penalty can cause coefficients to drop to zero unpredictably due to the stochastic nature of updates, creating jumps in the MSE. Ridge regression can also show instability because the regularization continuously adjusts the magnitudes of the coefficients.

2.4 Report

Differences between Gradient Descent Algorithms:
Gradient Descent is an optimization algorithm used to minimize a cost function by adjusting model parameters step by step. The goal is to move toward the lowest point of the function (the "global minimum") by following the slope or gradient.

1. **Batch Gradient Descent (BGD):**
Batch Gradient Descent computes the gradient of the cost function using the entire dataset for each step of the optimization process. It updates the parameters after evaluating the cost across all data points.

Advantages:

- Batch Gradient Descent is very precise because it uses the complete dataset to calculate the gradient, ensuring that each step is accurate and well-informed.
- It works well for small datasets, where the computational cost of processing the entire dataset is manageable.

Disadvantages:

- It is computationally expensive for large datasets since it requires processing the entire dataset before each update.
- It consumes a lot of memory because the algorithm must load and store the entire dataset at every step.

2. Stochastic Gradient Descent (SGD):
Stochastic Gradient Descent updates the model parameters after calculating the gradient using only one randomly selected data point from the dataset at each step. This randomness makes it faster but less precise compared to Batch Gradient Descent.

Advantages:

- Stochastic Gradient Descent is faster than Batch Gradient Descent because it does not need to process the entire dataset for each step.
- The randomness allows it to escape local minima (small dips in the cost function that are not the global minimum), which helps in finding better solutions.

Disadvantages:

- It is noisy and less stable because using a single data point introduces randomness into the optimization process. This can lead to a bumpy path toward the minimum.

- It may take longer to converge to the minimum, as the updates fluctuate due to the single data point's variability.

3. Mini-Batch Gradient Descent (MBGD):
Mini-Batch Gradient Descent divides the dataset into smaller groups called mini-batches. The gradient is calculated using a mini-batch, and the parameters are updated after each batch.

Advantages:

- Mini-Batch Gradient Descent is faster than Batch Gradient Descent because it processes smaller chunks of data instead of the entire dataset.
- It reduces the noise seen in Stochastic Gradient Descent, leading to smoother updates and better convergence.
- It is highly efficient on modern hardware, like GPUs, which can process mini-batches in parallel.

Disadvantages:

- The choice of mini-batch size is crucial. Too small, and it behaves like SGD (noisy); too large, and it approaches the computational cost of BGD.
- It may still require careful tuning to achieve optimal performance.

Gradient descent method that converged the fastest:

The Mini-batch Gradient Descent (MBGD) method converged the fastest because it reached a near-optimal MSE within just 500 iterations. On the other hand, Batch Gradient Descent (BGD) required 1000 iterations to achieve a slightly better MSE . While BGD achieved a marginally lower error, the faster convergence of MBGD in terms of iterations makes it the quickest method in this case.

How Lasso and Ridge Regularization Influence the Model:

Lasso regularization, also known as L1 regularization, modifies the loss function by adding a penalty proportional to the absolute values of the model’s coefficients. This technique encourages sparsity by driving some coefficients to exactly zero, effectively filtering out less important features.

As a result, Lasso is especially valuable when dealing with datasets that have a large number of features, as it not only simplifies the model but also reduces the risk of overfitting by focusing on the most relevant variables.

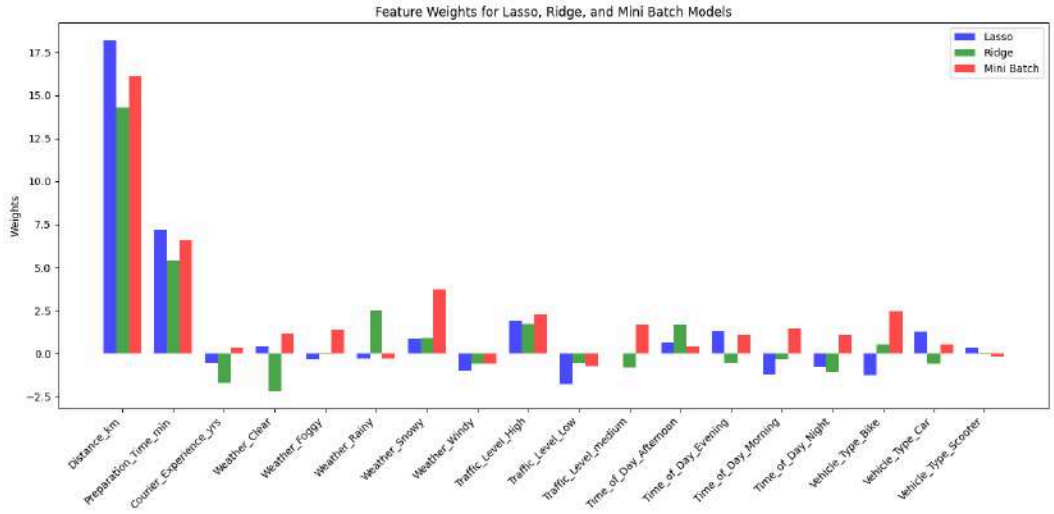
On the other hand, Ridge regularization, or L2 regularization, introduces a penalty based on the squared values of the coefficients. While Ridge also reduces the magnitude of the coefficients, it does not shrink them to zero entirely. This characteristic ensures that all features contribute to the model, making it particularly useful in cases of multicollinearity, where features are highly correlated. By evenly distributing the influence across all features, Ridge regularization enhances the stability of the model and helps prevent overfitting.

The optimal lambda (amongst the ones I have chosen) for Lasso and Ridge based on test performance is 0.01

Scaling of features - model performance

Scaling features ensures that all features contribute equally to the model. Without scaling, features with larger numerical ranges dominate, leading to biased models, especially in algorithms that rely on distance calculations (like KNN or SVM) or gradient-based methods (like Linear Regression). Scaling helps improve model performance by ensuring fair comparison between features, accelerating convergence in optimization, and leading to more accurate results.

Bar Plot - Lasso, Ridge and Mini-Batch:



Distance_km and Preparation_time affect the output the most (have positive correlation with target variable) . Negative weights in these methods usually occur because the corresponding features have a negative correlation with the target variable. In other words, as the feature value increases, the predicted outcome decreases.
If the magnitude of weights is compared then the mini-batch has comparatively higher magnitude as no regularization is done.

Order_ID being unique for each delivery does not contribute to the evaluation of output i.e, delivery time. Along with this, foggy weather and scooter as vehicle type have very less effect.

3 KNN and ANN

3.1 KNN

Task 1: Classification

- Classify each image in the test set using the labels of the k nearest neighbours from the train set. Report the accuracy for 3 different values of k(1, 5, 10) and cosine and euclidean distance metrics

```
Metric: euclidean
Testing with k = 1
Accuracy: 90.47999978065491%
Testing with k = 5
Accuracy: 91.82000160217285%
Testing with k = 10
Accuracy: 91.93999767303467%

Metric: cosine
Testing with k = 1
Accuracy: 90.47999978065491%
Testing with k = 5
Accuracy: 91.80999994277954%
Testing with k = 10
Accuracy: 91.93999767303467%
```

- Using k = 1, get the text embedding closest to each image and predict the accuracy

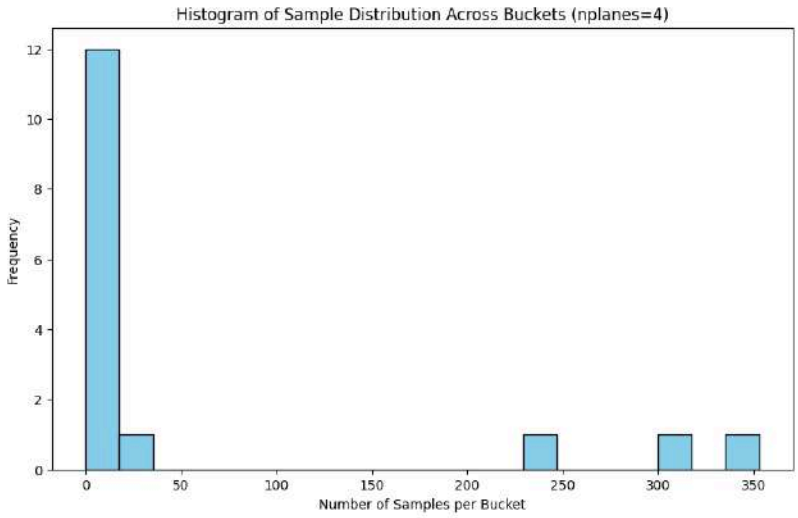
```
Accuracy with text embeddings: 87.80999755859375%
```

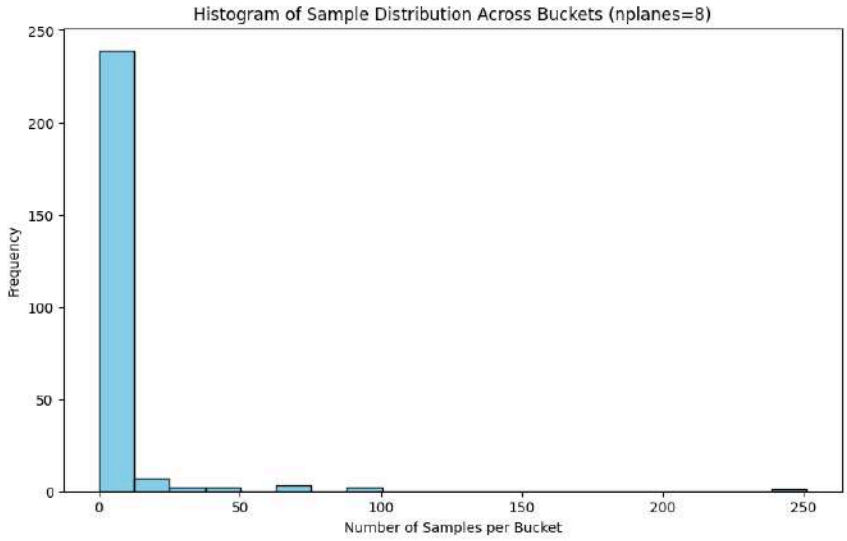
Task 2: Retrieval

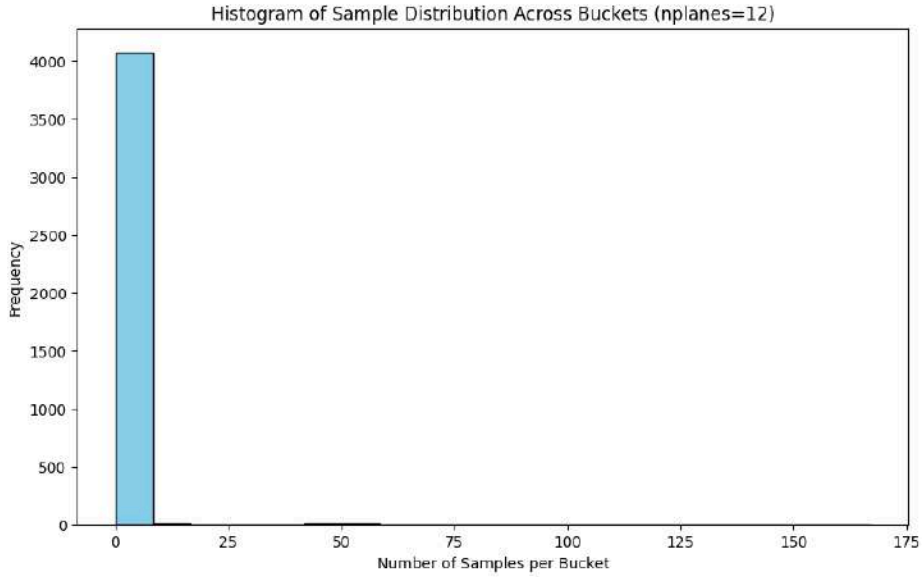
```
Text to Image Retrieval Metrics: {'mean_reciprocal_rank': 1.0, 'precision_at_k': 0.974, 'hit_rate': 1.0}
Image to Image Retrieval Metrics: {'mean_reciprocal_rank': 0.1179105452771932 7, 'precision_at_k': 0.100355, 'hit_rate': 0.293}
```

3.2 Locally Sensitive Hashing

Histograms showing frequency of samples in each bucket: nplanes=4,8,12







Problems arised:

1.Buckets are unevenly filled: Some buckets have way too many samples, while others are almost empty or completely unused.This uneven distribution can make LSH less effective since dense buckets force you to do extra comparisons, while empty ones add no value.

2. Computational and Memory Overhead: As the number of hyperplanes increases, the theoretical number of buckets grows exponentially, but in practice, only a few buckets are utilized. Random data distribution may not align well with the hyperplanes, causing uneven bucket populations.

Image to Image retrieval : k=100

```
Mean Reciprocal Rank (MRR): 0.6692514381880846
Precision@100: 0.6740999999999999
Hit Rate: 0.999
```

Changing the number of hyperplanes:

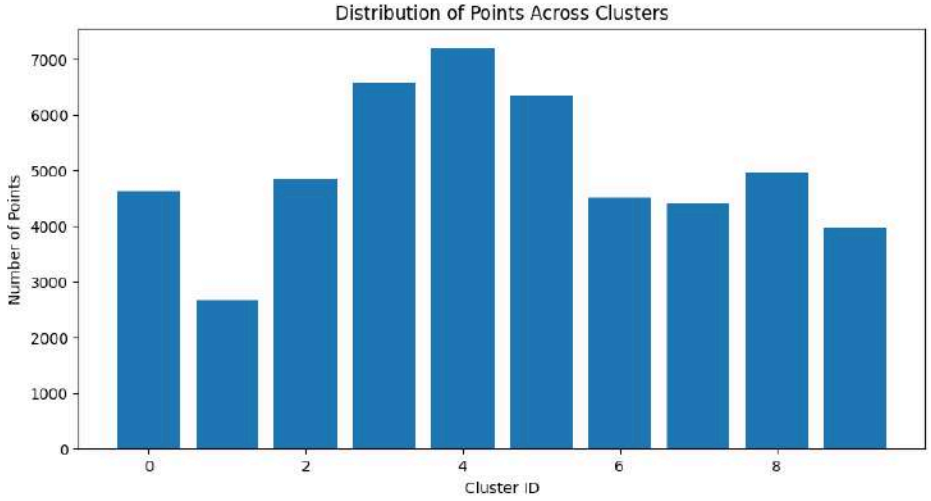
```
Results for 8 hyperplanes:
Mean Reciprocal Rank (MRR): 0.7213163789350423
Precision@100: 0.7151430000000001
Hit Rate: 0.9988

Results for 12 hyperplanes:
Mean Reciprocal Rank (MRR): 0.7081812198187221
Precision@100: 0.704511
Hit Rate: 0.9982

Results for 16 hyperplanes:
Mean Reciprocal Rank (MRR): 0.4944585769473964
Precision@100: 0.552036
Hit Rate: 0.9983

Results for 20 hyperplanes:
Mean Reciprocal Rank (MRR): 0.08291677855068474
Precision@100: 0.20889300000000002
Hit Rate: 0.999
```

Observations:

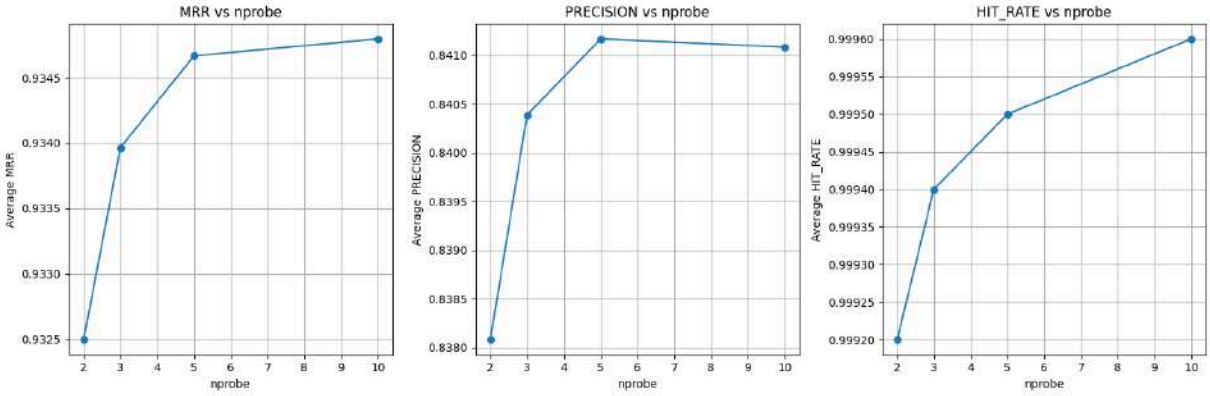
1. Mean Reciprocal Rank (MRR):
    - Decreases as the number of hyperplanes increases.
    - Likely because more hyperplanes make the hashing too specific, splitting similar samples into separate buckets and reducing retrieval accuracy.
2. Precision@100:
    - Drops with more hyperplanes.
    - This happens because fewer relevant samples are grouped together in the top results due to increased sparsity in buckets.
3. Hit Rate:
    - Remains stable but shows a slight decline with more hyperplanes.
    - Indicates that most relevant samples are still found, but a few may be lost due to overly sparse or inefficient bucket usage.

3.3 IVF

Task 1:



Observation: The number of points per cluster is unevenly distributed. Some clusters are denser than others, which is common in high-dimensional data due to the non-uniform distribution of feature vectors.

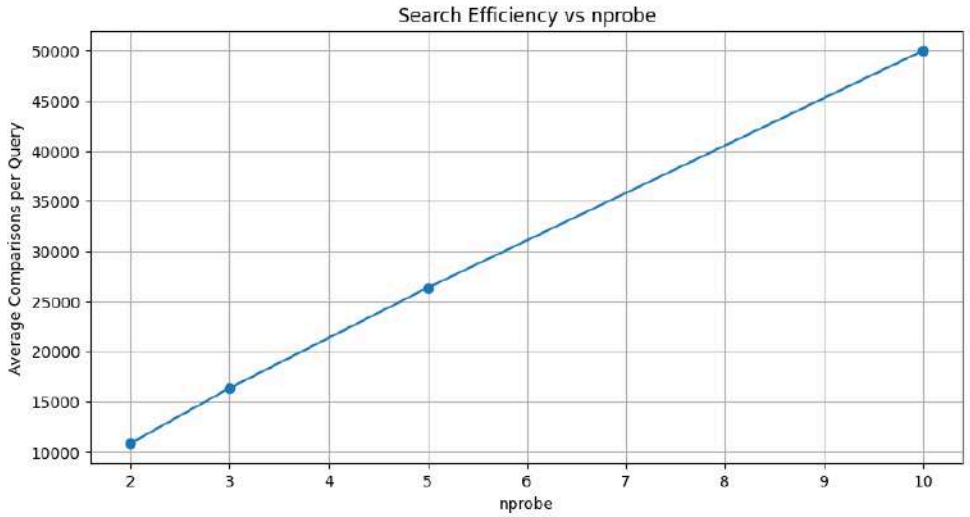


Observations:

As nprobe increases, more clusters are searched, which generally improves the Mean Reciprocal Rank (MRR). This is because expanding the search space increases the likelihood of finding the relevant image earlier in the ranking. However, this improvement is not linear; after a certain point, the MRR flattens since most relevant clusters have already been explored.

Hit Rate also increases with higher nprobe values, as probing more clusters raises the chances of including at least one relevant image in the retrieved set.

Similar to MRR, this effect diminishes at high nprobe values when almost all relevant clusters are being searched
Precision, on the other hand, tends to slightly decrease with increasing nprobe beyond a point. This is because searching more clusters retrieves a larger set of images, which may include more irrelevant ones, thereby diluting precision.

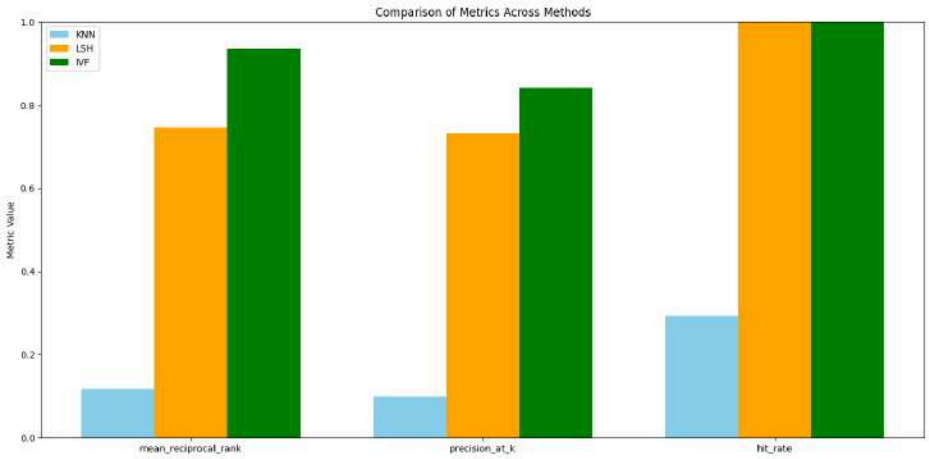


Observations: As nprobe increases, the average number of comparisons also increases, since more clusters are searched.This improves recall (i.e., more relevant images are found) but also increases computation time.Conversely, lower nprobe values result in fewer comparisons, leading to faster retrieval but potentially lower accuracy due to missed relevant clusters.

```
Average Metrics:
nprobe  MRR     Precision@100   Hit Rate        Comparisons
------------------------------------------------------------
2       0.9325  0.8381  0.9992  10817.4
3       0.9340  0.8404  0.9994  16327.1
5       0.9347  0.8412  0.9995  26332.6
10      0.9348  0.8411  0.9996  50000.0
```

3.4 Analysis

Comparing three methods - based on performance:



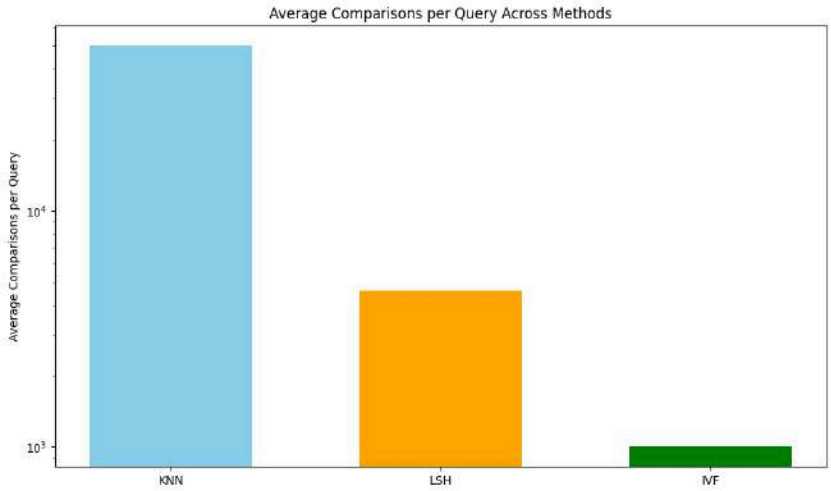
Observations: It can be observed that IVF performs the best followed by LSH and then KNN.

KNN's poor performance could be due to the high-dimensionality of the data, where distances become less meaningful, leading to inaccurate neighbor identification(curse of dimensionality). This issue is common in high-dimensional spaces because all points tend to be nearly equidistant, making it challenging to find true nearest neighbors.

LSH performed better than KNN because it groups similar items into hash buckets, which helps in reducing the noise from irrelevant dimensions. This method is effective in high-dimensional spaces as the hash functions are chosen to capture the data's similarity patterns.

IVF achieved the highest accuracy as it efficiently clusters the dataset into meaningful partitions, enabling more focused and relevant searches. By probing multiple clusters (with nprobe), IVF effectively balances recall and precision

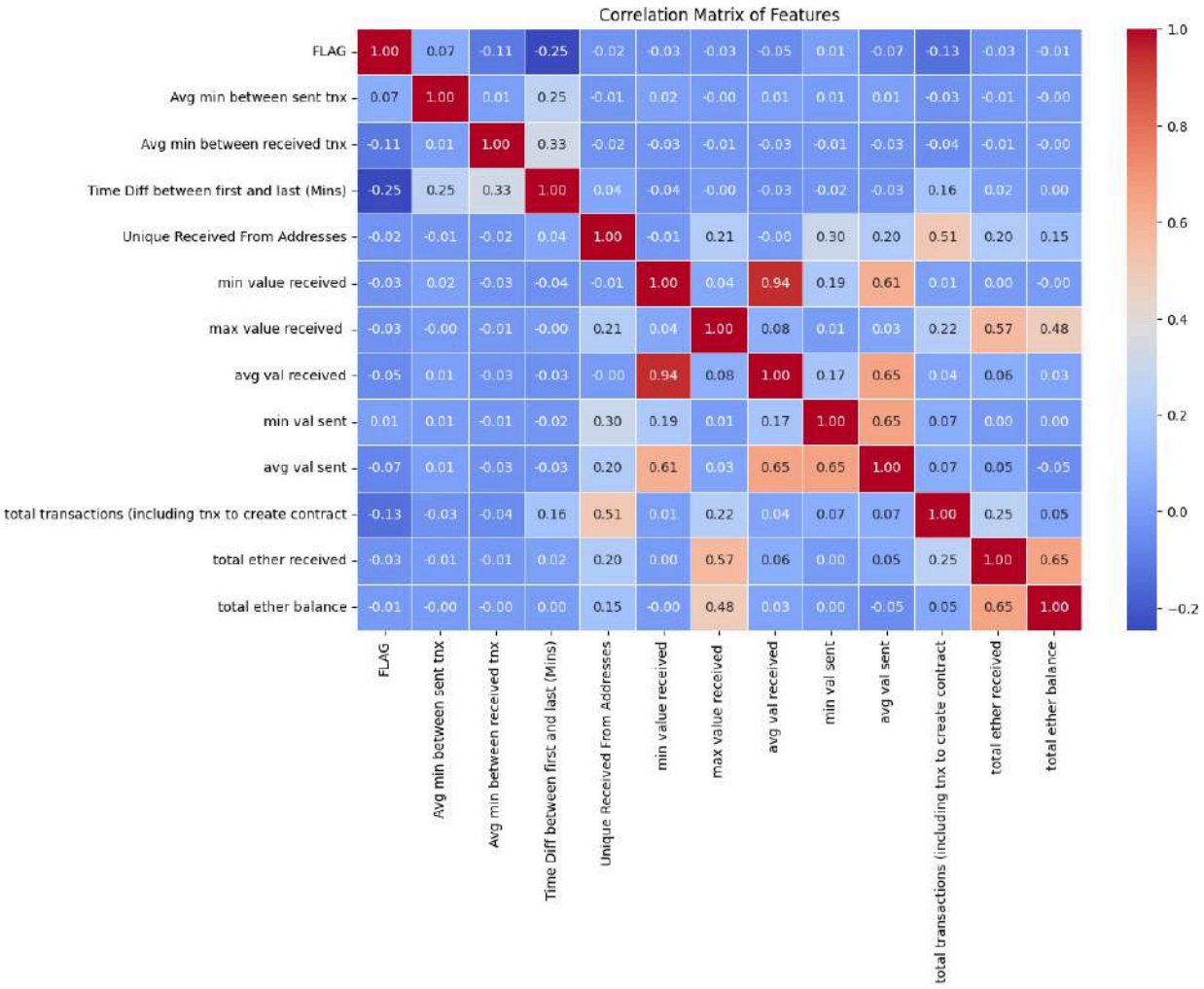
Comparing three methods - based on average number of comparisons per query:



It can be observed that KNN takes maximum comparisons per query followed by LSH and then IVF.
KNN performs an exhaustive search, comparing the query point with every point in the dataset. This brute-force approach ensures the most accurate neighbors but requires the maximum number of comparisons, making it computationally expensive.
LSH required more comparisons per query than IVF because the hash functions used may not have effectively grouped similar points together due to complex data distribution.
IVF reduces the number of comparisons by first assigning data points to clusters and then only searching within a subset of clusters using the nprobe parameter.

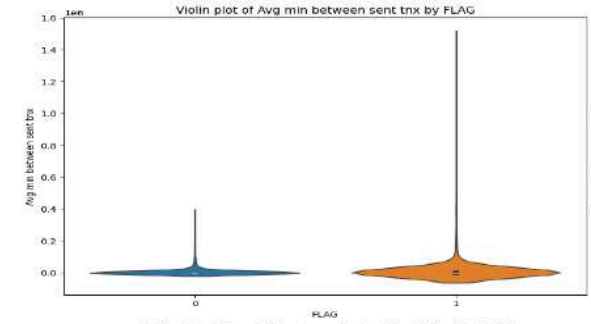
4 The Crypto Detective

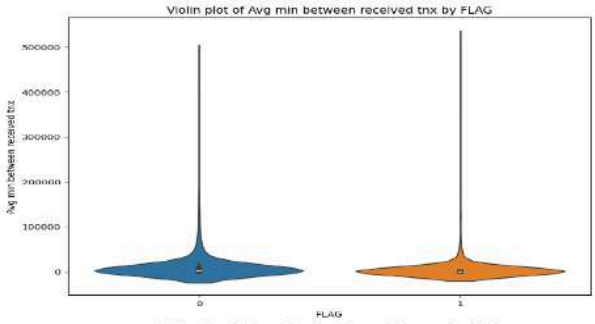
4.1 Preprocessing and Exploratory Data Analysis

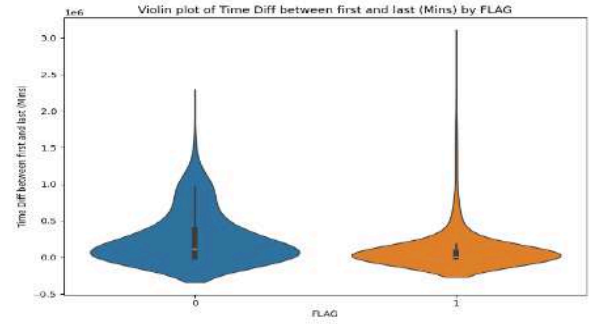


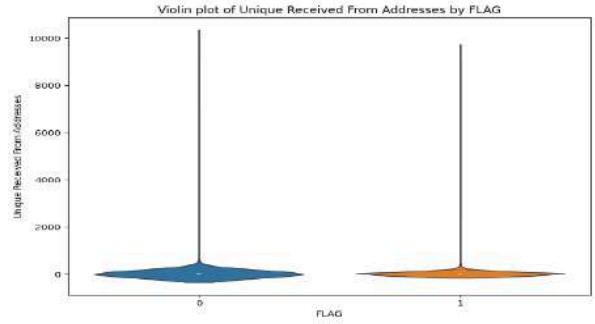
Observations:

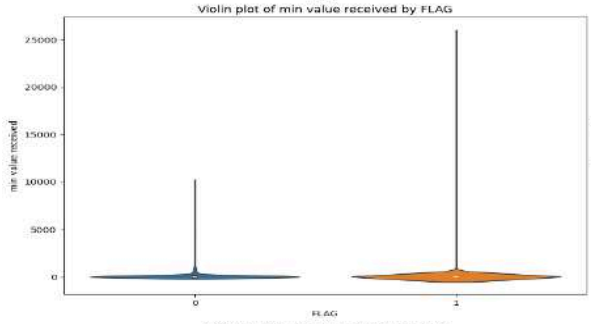
- **avg min between sent tnx** and **min val sent** show a positive correlation with **FLAG**, indicating that as the average minimum time between sent transactions or the minimum value sent increases, the likelihood of the target outcome (FLAG) being positive also increases. This suggests that users who make less frequent transactions or send smaller amounts are more likely to be flagged.
- The rest of the features, including metrics like **total ether received**, **total ether balance**, and other transaction-related features, are negatively correlated with **FLAG**. This indicates that higher transaction volumes, larger balances, or increased activity are associated with a lower probability of being flagged. This could imply that more active or wealthier accounts are less likely to trigger the flag condition.
- **avg and min value received** have a strong positive correlation, suggesting that accounts receiving higher average values also receive high minimum values. This consistency might indicate a pattern in receiving transactions, possibly from regular or consistent sources.Similarly, **total ether received** and **total ether balance** are strongly correlated. This is logical because the total balance is influenced by the total ether received minus the total ether sent. Accounts that receive more ether are likely to have higher balances.

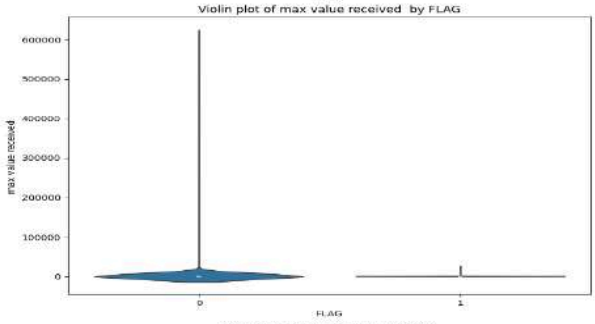

Violin plot of Avg min between sent tnx by FLAG
1e6
Avg min between sent tnx
FLAG

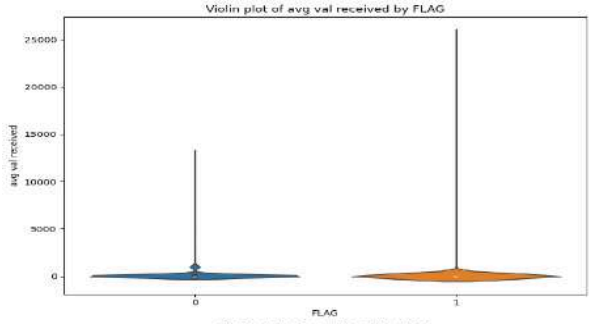

Violin plot of Avg min between received tnx by FLAG
Avg min between received tnx
FLAG

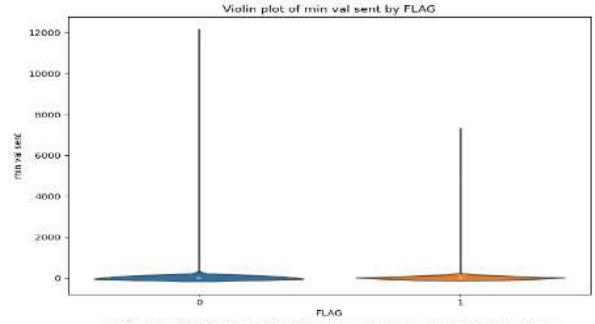

Violin plot of Time Diff between first and last (Mins) by FLAG
1e6
Time Diff between first and last (Mins)
FLAG

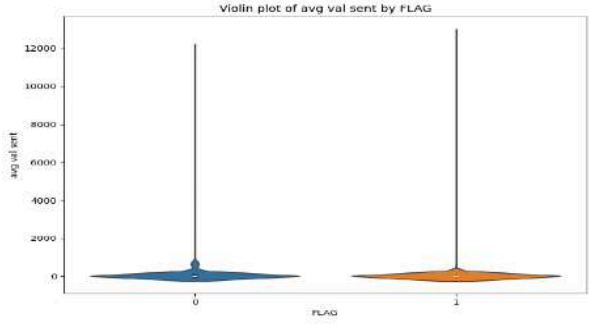

Violin plot of Unique Received From Addresses by FLAG
Unique Received From Addresses
FLAG

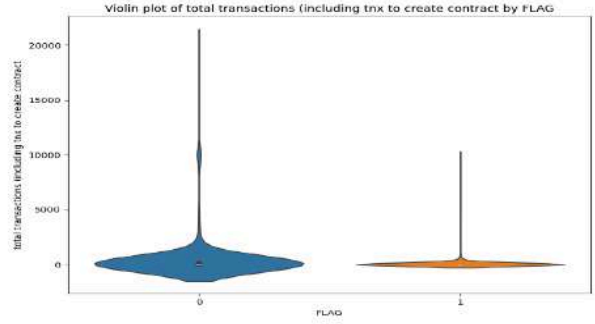

Violin plot of min value received by FLAG
min value received
FLAG

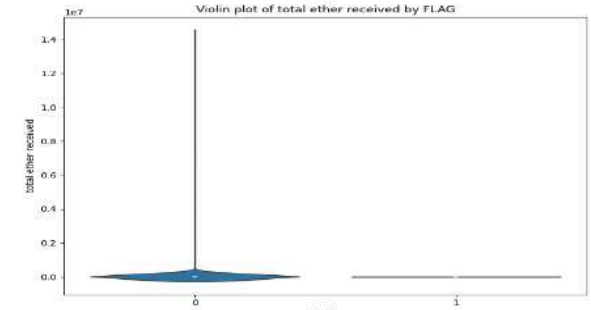

Violin plot of max value received by FLAG
max value received
FLAG

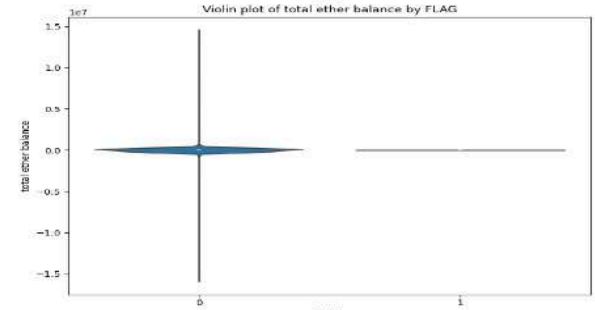

Violin plot of avg val received by FLAG
avg val received
FLAG


Violin plot of min val sent by FLAG
min val sent
FLAG


Violin plot of avg val sent by FLAG
avg val sent
FLAG


Violin plot of total transactions (including tnx to create contract by FLAG
total transactions (including tnx to create contract)
FLAG


Violin plot of total ether received by FLAG
1e7
total ether received
FLAG


Violin plot of total ether balance by FLAG
1e7
total ether balance
FLAG

Observations:

- The columns **max value received**, **total ether received**, and **total ether balance** show no significant variation between FLAG = 0 and FLAG = 1. This suggests that the occurrence of FLAG = 1 is not influenced by these features, indicating that flagged accounts have similar distributions of maximum received value, total ether received, and total balance as non-flagged accounts.
- All the violin plots are asymmetric, showing that the data distributions are skewed. This asymmetry suggests that most data points are concentrated towards one side, with a long tail on the other side, indicating the presence of outliers or extreme values. This could be due to a few accounts with very high transaction values or very frequent transactions.
- The plots show a thin stretch between the minimum and maximum values, revealing that the majority of data points are concentrated in a narrow range. This indicates low variance and suggests that most accounts exhibit similar behavior for these features, with only a few outliers having extreme values.

4.2 Fraud Detection Model

Accuracy Comparison:

```
Custom Decision Tree - Accuracy on the test data: 0.8520
Custom Decision Tree - Accuracy on the validation data: 0.8446
Training time for Custom Decision Tree model: 71.1086 seconds


Scikit-learn Decision Tree - Accuracy on the test data: 0.8520
Scikit-learn Decision Tree - Accuracy on the validation data: 0.8450
Training time for Scikit-learn model: 0.1075 seconds

Comparison:
Accuracy Difference (Test): 0.0000
Accuracy Difference (Validation): 0.0004
Training Time Difference: -71.0012 seconds
```

- **Test Accuracy**: Both custom decision tree and the Scikit-learn decision tree achieved the same accuracy on the test set (0.8520).
- **Validation Accuracy**: There is a very small difference in validation accuracy, with Scikit-learn performing slightly better (0.8450) compared to your custom model (0.8446).

Observations:
The training time for custom decision tree is much higher than for Scikit-learn. The Scikit-learn implementation is highly optimized and utilizes efficient algorithms, parallel processing, and hardware acceleration (such as multi-threading) that custom implementation might lack.

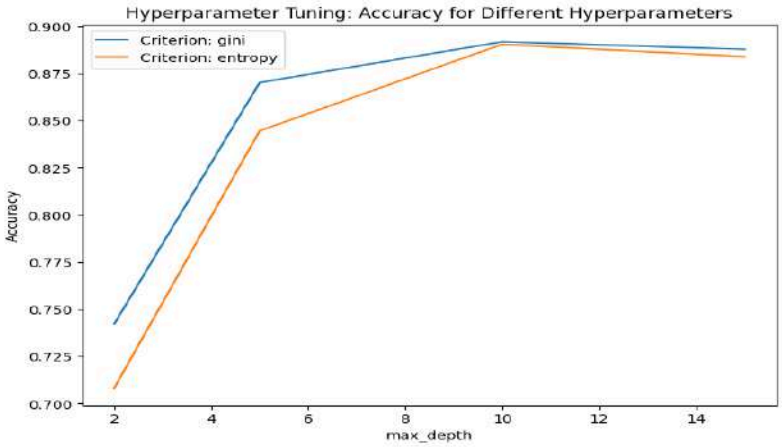
4.3 Hyperparameter Tuning

```
Best hyperparameters:
max_depth               10.0
min_samples_split          5
criterion               gini
accuracy            0.891832
Name: 14, dtype: object
```

Observations:

- When the tree is shallow (low max_depth), it's too simple to capture the complexity of the data, so both training and validation accuracy are low. As you increase the depth, the tree gets better at finding patterns in the data, and the accuracy improves for both training and validation.
- At a certain point, the validation accuracy peaks— where the tree is just complex enough to capture the important patterns without overfitting. Beyond this, as the tree gets even deeper, it starts overfitting the training data. The training accuracy might continue to increase, but the validation accuracy begins to drop because the tree is now memorizing the training data instead of generalizing to new, unseen data.

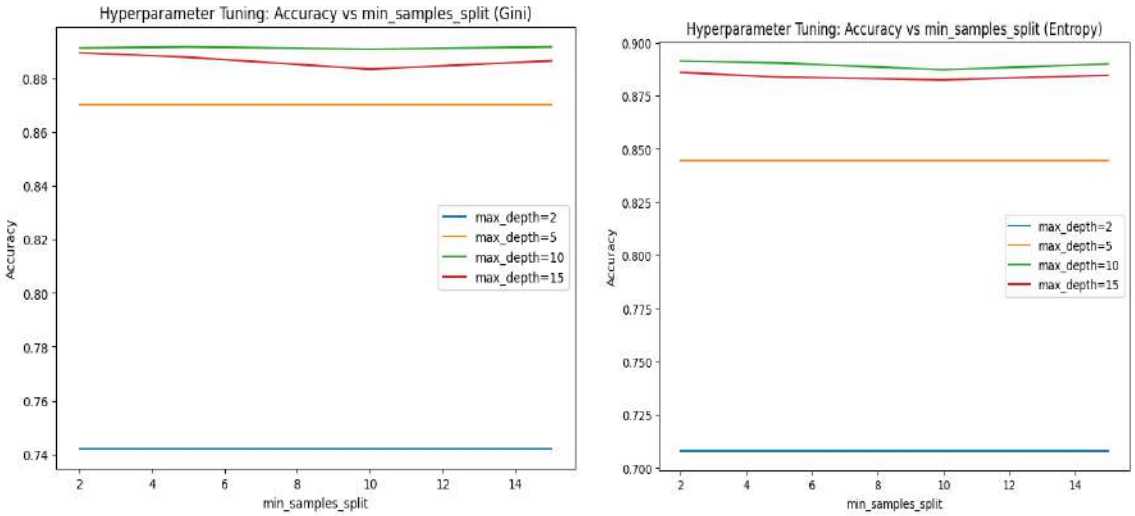
Accuracy vs max_depth:



Observation:

- As the max_depth of the tree increases, the accuracy initially rises sharply because the model becomes more capable of capturing complex patterns in the data. This steep increase continues up to max_depth = 5,where it predicts outcomes with better precision. After this point, the accuracy continues to improve but at a slower rate, showing a more gradual increase as the tree fine-tunes its decision boundaries.
- The accuracy reaches its peak at max_depth = 10, where the tree is complex enough to capture most of the relevant patterns without becoming overly specific. However, beyond this point, a slight decrease in accuracy is observed, indicating that the tree has started to overfit.

Accuray vs min_samples_split:

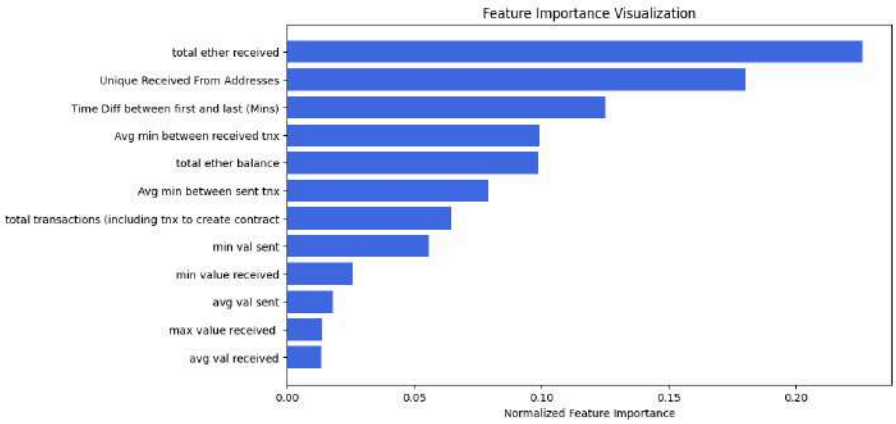


Observation:

- For low values of max_depth, the choice of min_samples_split does not significantly affect accuracy because the tree is shallow and not complex enough to benefit from variations.
- However, for max_depth values above 10, a slight decrease in accuracy is observed when min_samples_split is set above 10. This happens because a higher min_samples_split value restricts the tree from growing further by requiring more samples at each node before a split can occur. As a result, the tree becomes less flexible and may miss important patterns, leading to a small decline in accuracy.

4.4 Model Visualization

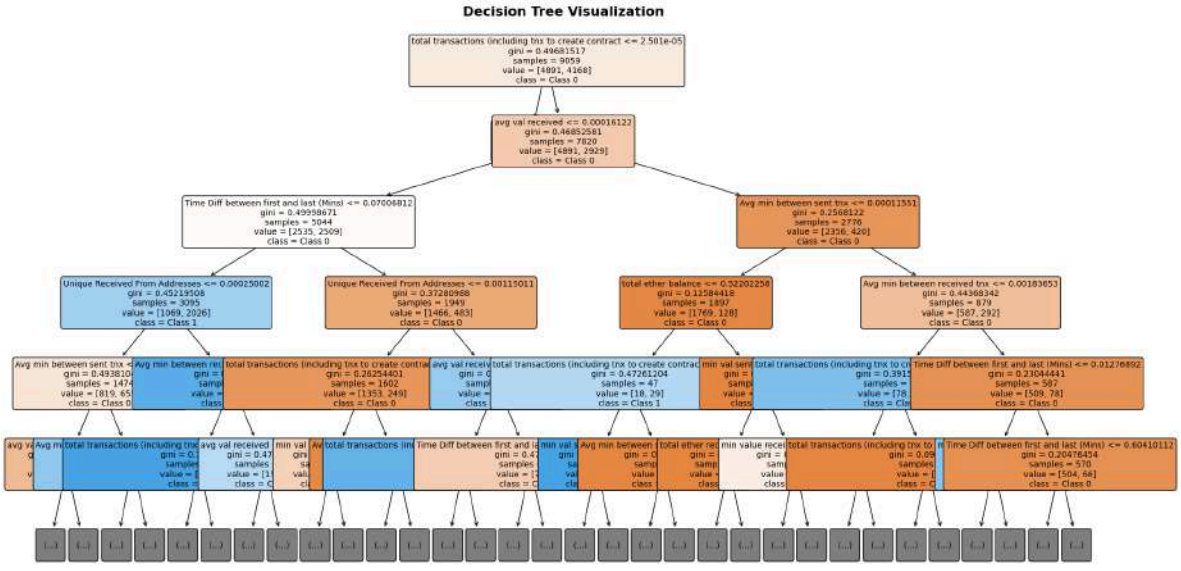
Feature Importance Visualization:



Observation:

- The feature that contributes the most to the model's performance is the total ether received. This suggests that the total amount of ether accumulated is highly influential in predicting the output variable.
- The least contributing features are max value received and avg val received. This indicates that individual transaction values, whether maximum or average, are not as impactful.Overall, the model seems to prioritize cumulative and network-related features over individual transaction statistics.

Tree Visualization:

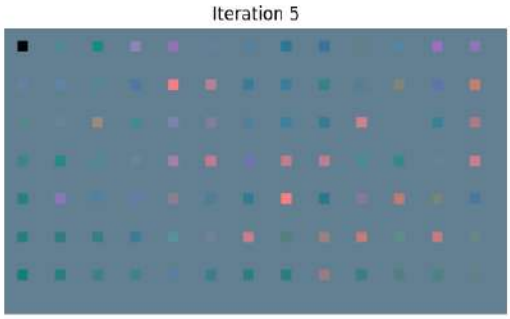


5 K-Means

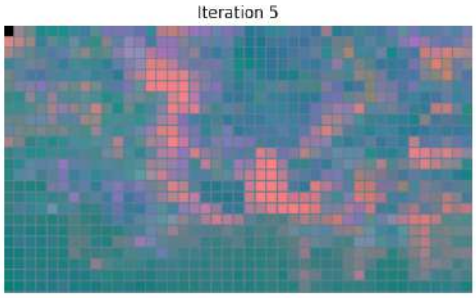
5.1 SLIC

Changing Hyperparameters: No.of clusters and compactness

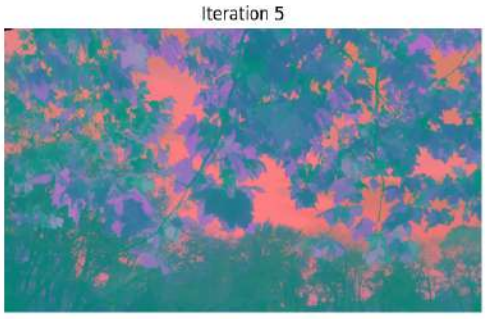
K = 100, compactness = 20



K = 1000, compactness = 20

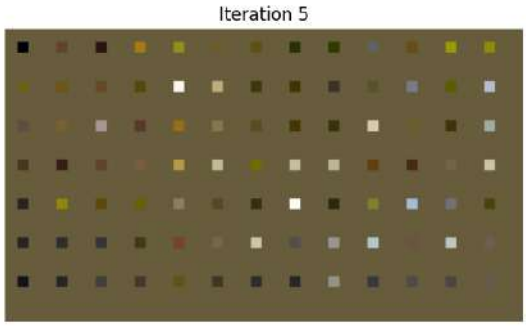


k=5000, compactness = 20

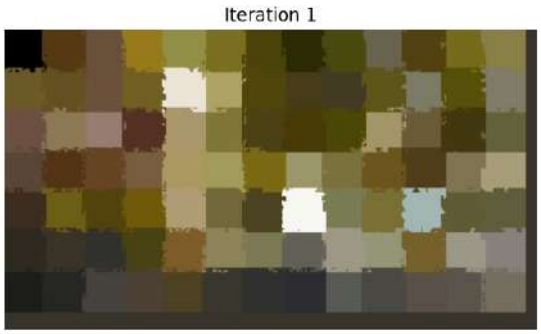


Distance metric : RGB space

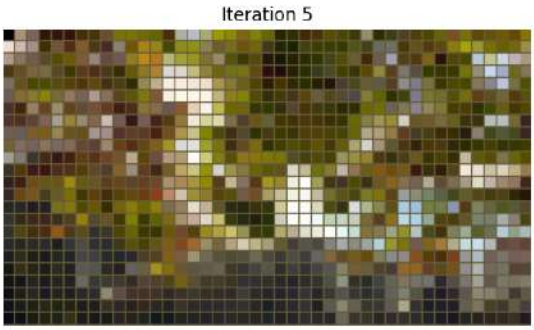
K = 100, compactness = 20



k = 100, compactness=80



K = 1000, compactness = 20



K = 1000, compactness = 80

K = 5000, compactness = 20

K = 5000, compactness = 80

Observations:

- Increasing the number of clusters results in finer segments, with more superpixels capturing detailed features and boundaries. This leads to more accurate edge preservation.Decreasing the number of clusters produces larger superpixels, simplifying the image representation.
- Higher compactness values prioritize spatial proximity over color similarity, resulting in more regular, grid-like superpixels. This can lead to less

accurate object boundaries but more uniform superpixel shapes.Lower compactness values emphasize color similarity, allowing superpixels to adapt more closely to object boundaries.

In RGB space, each pixel is represented by three channels—Red, Green, and Blue—each ranging from 0 to 255. These values correspond directly to screen colors, preserving the natural appearance of the image. As a result, the segmented output closely resembles the original image without noticeable color distortions.

In Lab space, the pixel values are represented by:

- L (Lightness): Ranges from 0 to 100, indicating brightness.
- a (Green-Red axis): Typically ranges from -128 to 127, representing color variations from green to red.
- b (Blue-Yellow axis): Also ranges from -128 to 127, capturing color differences from blue to yellow.

During segmentation in Lab space, the clustering is done based on perceptual differences.The color-opponent channels in Lab space don't directly map to the RGB primary colors, leading to unnatural hues in the output image.

5.2 Video Segmentation with SLIC

The frames segmented are stored in the folder named **processed_frames** and the video generated back from the video is **segmented_output.mp4**

5.3 Optimizing the video segmentation

The optimised frames segmented are stored in the folder named **optimized_frames** and the video generated back from the video is **opt_segmented_output.mp4**

The optimization version first checks if the current frame is significantly different from the previous one. If the change is small, it simply reuses the previous segmentation instead of recalculating everything, which saves a lot of time.

When it does need to recalculate, it only examines pixels close to each cluster center rather than scanning the entire image. This localized approach reduces the workload.By placing the initial cluster centers evenly across the image, the algorithm starts off with a good approximation, leading to faster convergence.

Overall, it balances accuracy and efficiency by focusing its efforts only where needed.

Comparison of Average Iterations:

- Without Optimization: Every frame is segmented independently, leading to a consistent and high number of iterations per frame.
- With Optimization: The first frame takes the full number of iterations, but subsequent frames require significantly fewer iterations. As a result, the average number of iterations per frame is reduced. This average decreases further with increased temporal consistency (i.e., less movement or change between frames).