Statistical Methods in AI
Assignment-1

Mayank Mittal
2022101094

Spring 2025

Contents

1 Food Delivery Time Prediction

1.1 Exploratory Data Analysis

For the food delivery time prediction task, we performed comprehensive exploratory data analysis on the dataset. Here are our key findings:

1.1.1 Data Preprocessing

- Missing values were handled using mean imputation for numerical features and mode imputation for categorical features
- Categorical features were encoded using Label Encoding
- The dataset was split into 70:15:15 ratio for train, validation, and test sets
- Features were scaled using StandardScaler to ensure uniform scale across all features

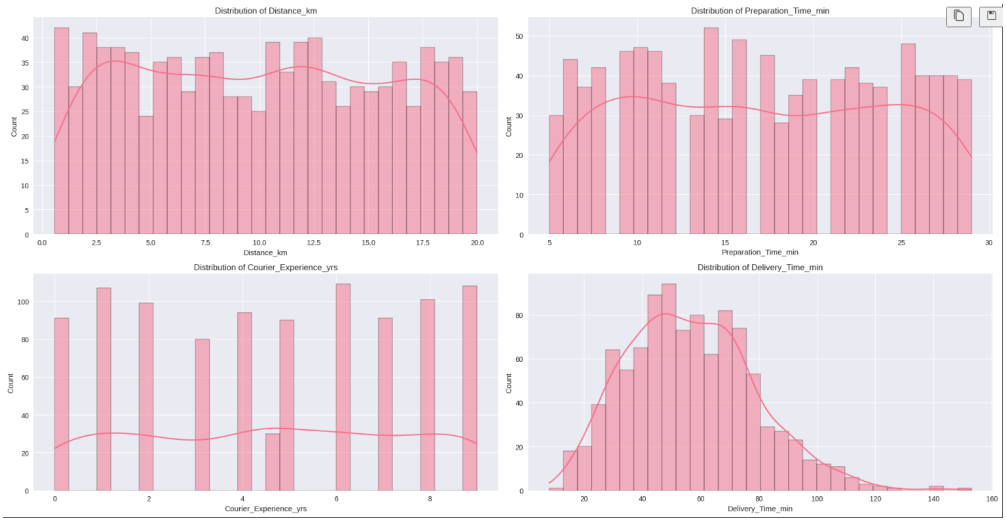

Figure 1: Histograms

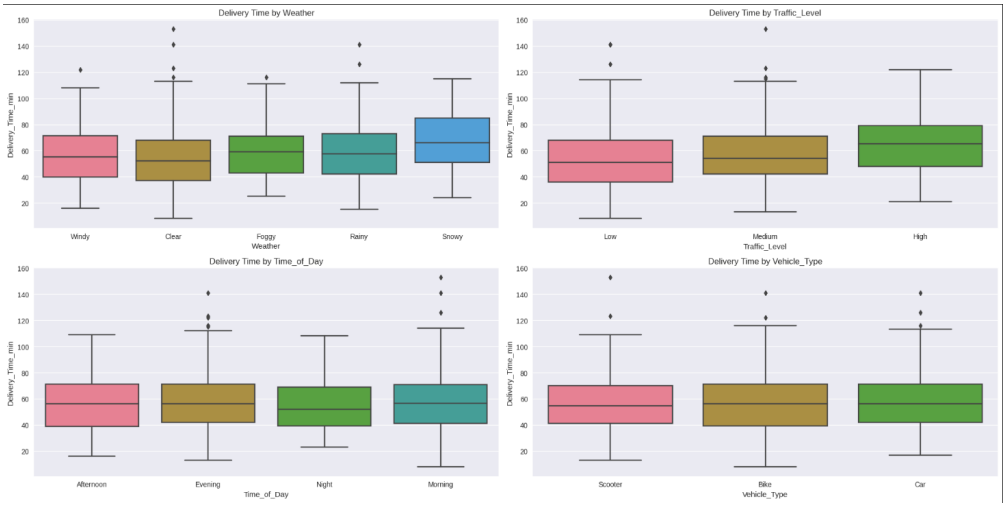

Figure 2: Box Plots

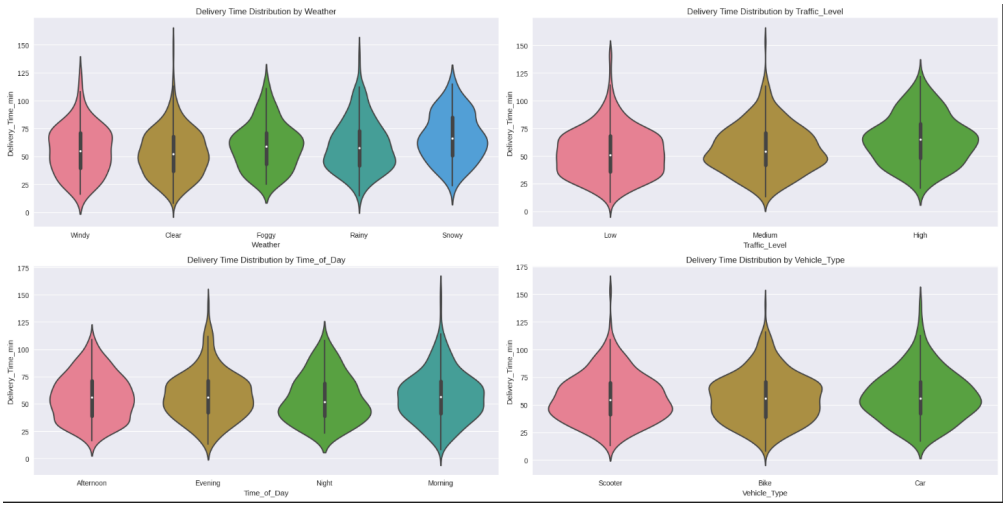

Figure 3: Violin Plots

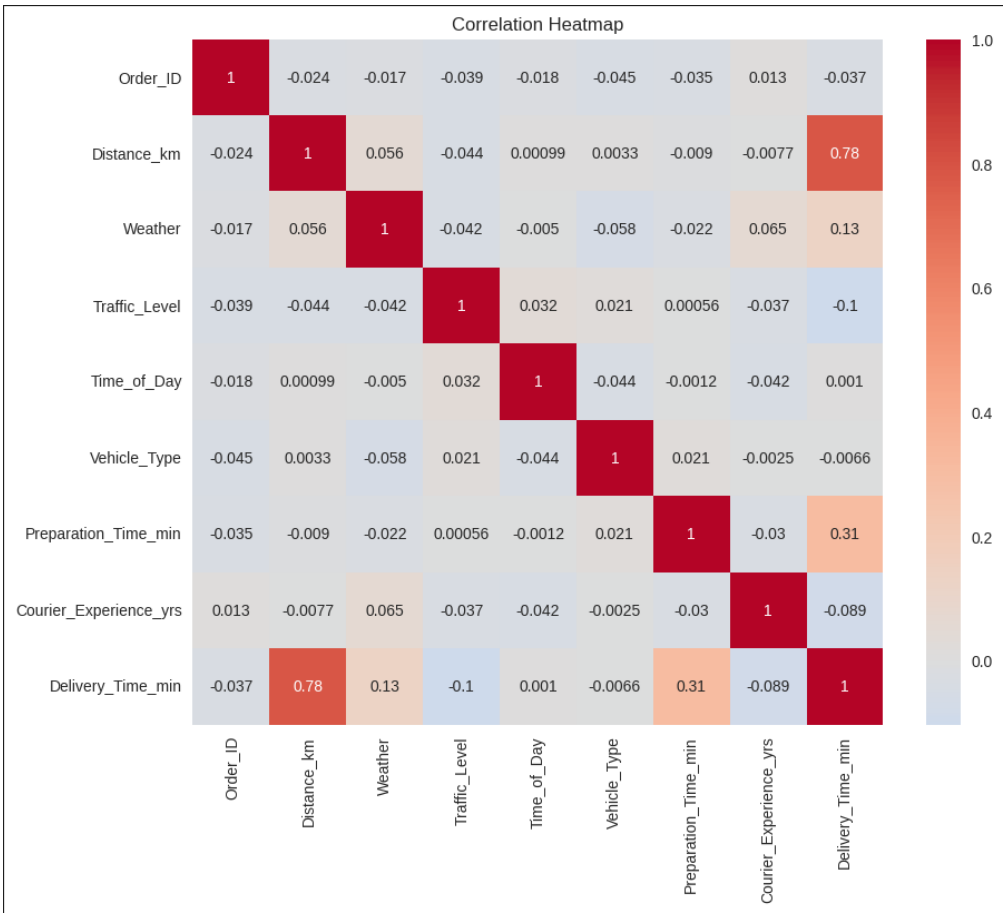

Figure 4: Correlation Heatmap

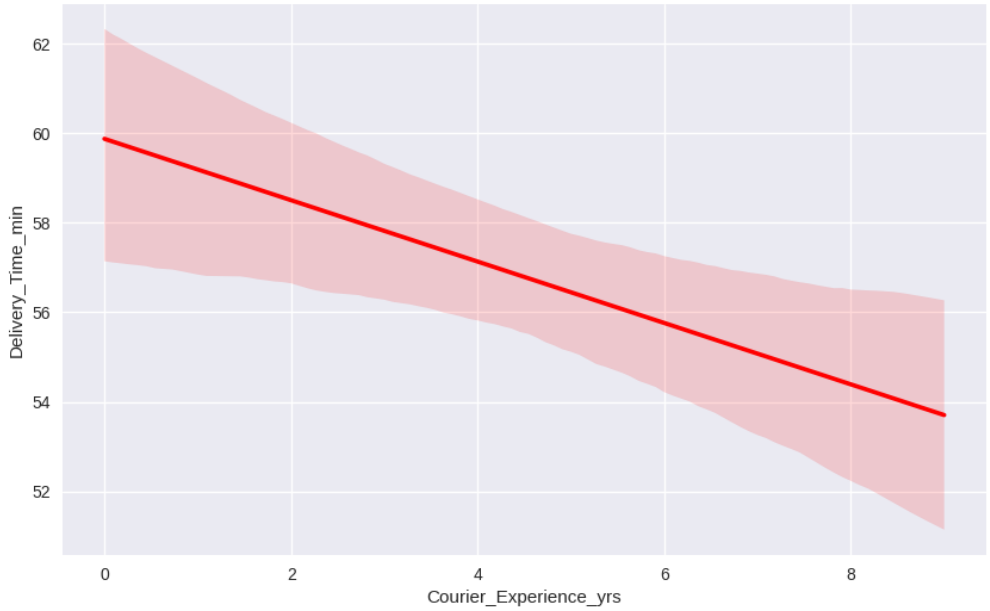

Figure 5: Line Plot

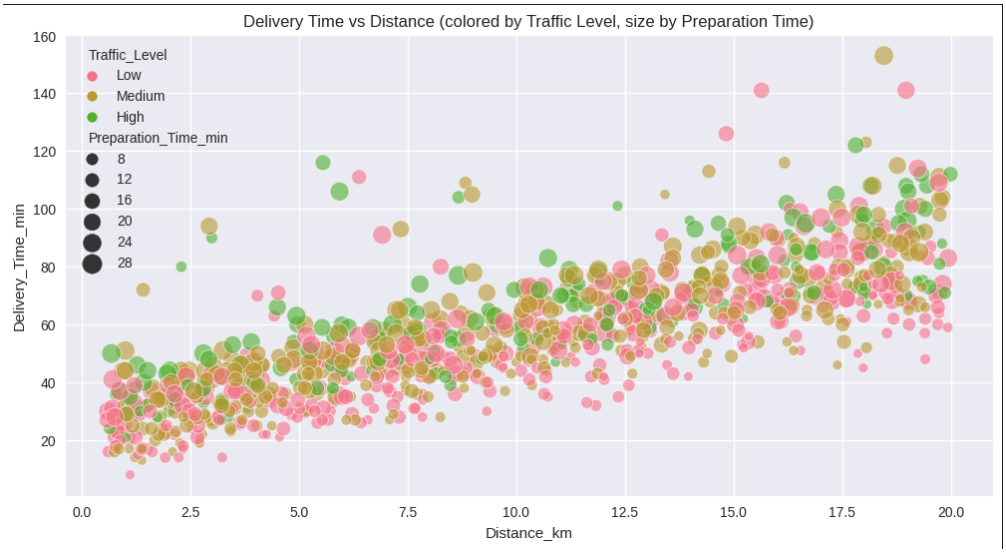

Figure 6: Scatter Bubble Plot

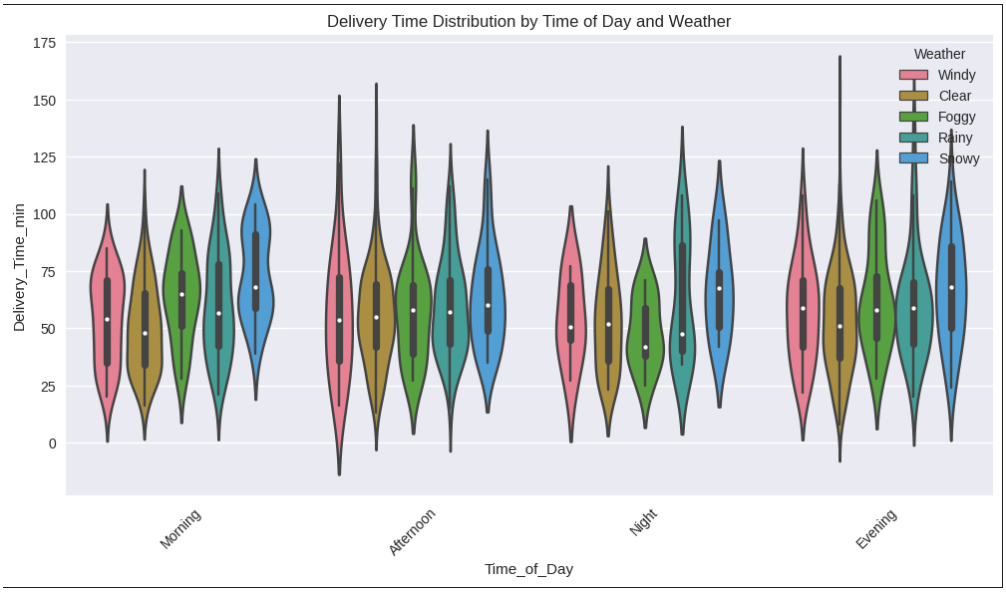

Figure 7: Violin Plot

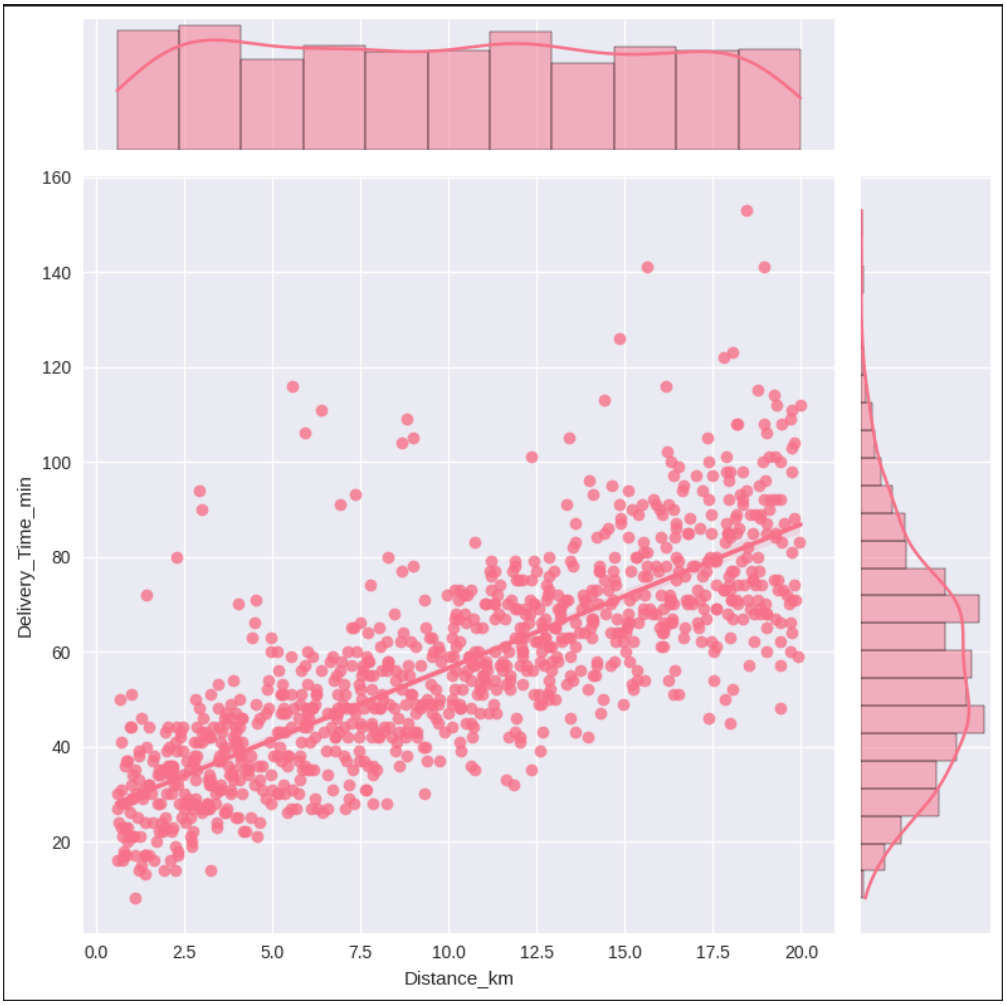

Figure 8: Joint Plot

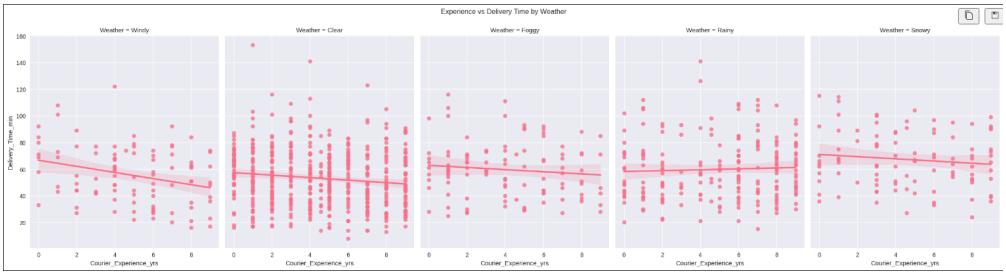

Figure 9: LMP plot

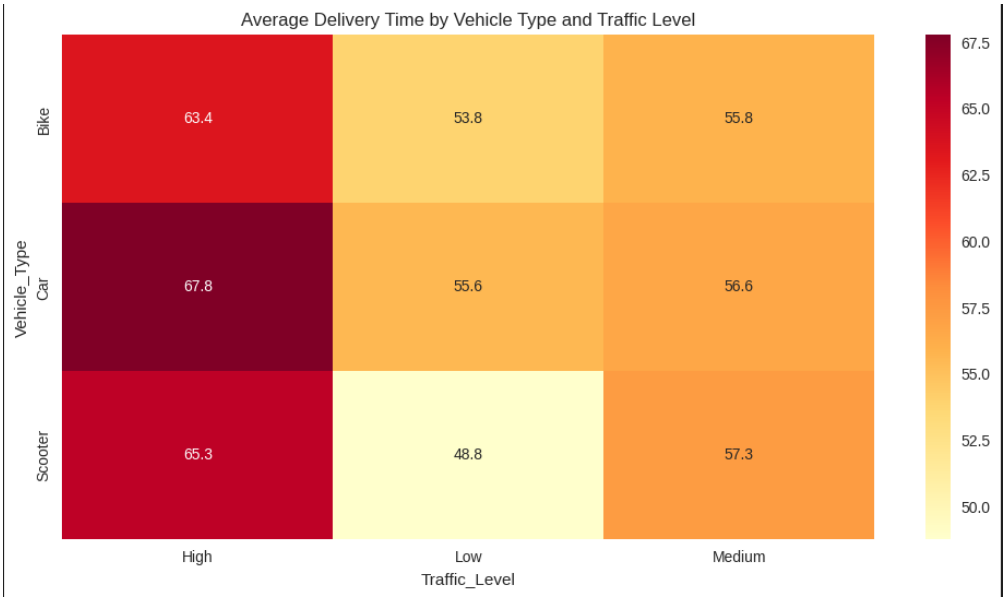

Figure 10: Heatmap

1.1.2 Key Observations

1. **Order Distribution:** Orders are uniformly distributed across all areas, with a typical range of 25-40.
2. **Preparation Time Clusters:** The histogram of `Preparation_time` reveals six distinct clusters, suggesting the presence of six different food categories with similar preparation times.
3. **Delivery Time Distribution:** The average and most probable delivery time is around **56 minutes**, as indicated by the distribution plot.
4. **Impact of Weather:** Median delivery time is highest in **Snowy** weather, followed by **Rainy, Foggy, and Windy** conditions, and is lowest in **Clear** weather.
5. **Effect of Traffic Levels:** Delivery time increases with traffic levels, being highest in **High Traffic**, followed by **Medium**, and lowest in **Low Traffic**.
6. **Time of Day and Vehicle Type:** No significant effect of `Time_of_day` and `Vehicle Type` on delivery time, as seen in the plots.
7. **Distance vs. Delivery Time:** A strong **positive correlation** exists between `Delivery_time` (minutes) and `Distance` (km), which is expected.
8. **Courier Experience:** A **negative correlation** is observed between `Courier Experience` (years) and delivery time, indicating that experienced couriers complete deliveries faster.
9. **Correlation Insights:** `Time_of_day`, `Courier_Experience`, and `Weather` show nearly zero correlation with `Delivery_time`.
10. **Vehicle Type and Traffic Levels:** From the heatmap, `Car` has the highest average delivery time in **High Traffic**, while `Scooter` has the lowest in **Low Traffic**, aligning with expectations.

1.2 Linear Regression with Gradient Descent

We implemented three variants of gradient descent:

1.2.1 Implementation Details

- Learning rate: 0.05
- Random weight initialization
- MSE loss function
- Maximum iterations: 1000

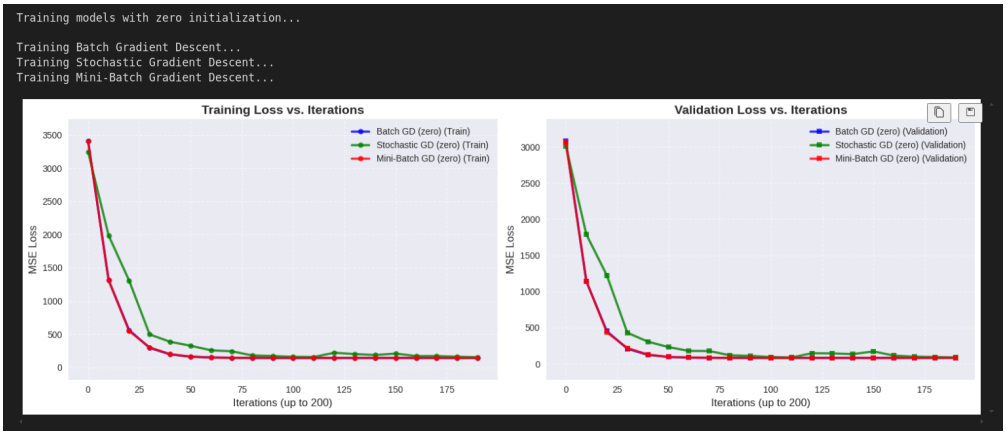

Figure 11: With zero initialisation

```
Model Performance Comparison:
                Method    Test MSE  R2 Score
0      Batch GD (zero)  137.624504  0.764858
1  Stochastic GD (zero)  147.684217  0.747670
2  Mini-Batch GD (zero)  139.343362  0.761921

Final Loss Values:
+----------------------+------------------+-----------------------+
|        Method        | Final Train Loss | Final Validation Loss |
+----------------------+------------------+-----------------------+
|   Batch GD (zero)    |    141.650614    |       82.333683       |
| Stochastic GD (zero) |    156.029047    |       95.640323       |
| Mini-Batch GD (zero) |    142.851123    |       85.110303       |
+----------------------+------------------+-----------------------+
```

Figure 12: Evaluation metric

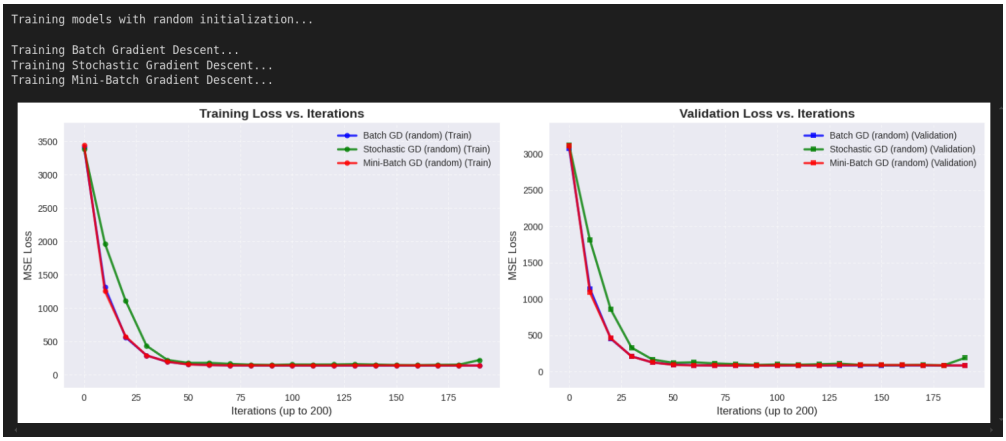

Figure 13: With random initialisation

```
Model Performance Comparison:
                   Method    Test MSE  R2 Score
0       Batch GD (random)  137.624504  0.764858
1  Stochastic GD (random)  158.471798  0.729238
2  Mini-Batch GD (random)  136.923237  0.766056

Final Loss Values:
+-----------------------+------------------+-----------------------+
|         Method        | Final Train Loss | Final Validation Loss |
+-----------------------+------------------+-----------------------+
|   Batch GD (random)   |    141.650614    |       82.333683       |
| Stochastic GD (random) |    159.957056    |       94.544841       |
| Mini-Batch GD (random) |    142.252791    |       83.524832       |
+-----------------------+------------------+-----------------------+
```

Figure 14: Evaluation metric

1. **Test MSE & R^2 Score:** Batch Gradient Descent (GD) performs identically for both zero and random initialization. However, Stochastic and Mini-Batch GD show slightly better performance with zero initialization, indicating more stable convergence.

2. **Final Train & Validation Loss:** Stochastic GD has significantly higher train and validation loss with random initialization, suggesting it struggles with convergence. Zero initialization yields lower losses, particularly for Stochastic and Mini-Batch GD.

3. **Overall Performance:** Batch GD remains consistent across both initializations, whereas Stochastic GD exhibits more instability with random initialization. Mini-Batch GD performs similarly in both cases but has slightly lower losses with zero initialization.

1.3 Regularization Analysis

For lambda = 0.5

```
Training Ridge Regression (λ=0.5)...

Training Lasso Regression (λ=0.5)...

Results with λ=0.5:
  Method    Test MSE  R2 Score
0  Ridge  315.119703  0.524891
1  Lasso  158.823382  0.752765

Performing lambda parameter tuning...

Regularization Impact on Test MSE and R² Score:
+--------+---------------+----------+---------------+----------+
| Lambda | Ridge Test MSE | Ridge R² | Lasso Test MSE | Lasso R² |
+--------+---------------+----------+---------------+----------+
|  0.0   |    140.3529    |  0.7602  |    146.3834    |  0.7499  |
|  0.11  |    189.3638    |  0.7165  |    143.1043    |  0.7613  |
|  0.22  |    233.8935    |  0.6575  |    147.9226    |  0.7587  |
|  0.33  |    264.4546    |  0.6169  |    152.402     |  0.7555  |
|  0.44  |    305.3326    |  0.5372  |    151.022     |  0.7638  |
|  0.56  |    323.9443    |  0.5276  |    158.8091    |  0.7562  |
|  0.67  |    345.5555    |  0.474   |    163.2877    |  0.7526  |
|  0.78  |    365.2238    |  0.4527  |    169.4272    |  0.7482  |
|  0.89  |    351.4142    |  0.4997  |    181.7933    |  0.7264  |
|  1.0   |    415.5814    |  0.3395  |    171.389     |  0.7632  |
+--------+---------------+----------+---------------+----------+
```

Figure 15: Performance metrics

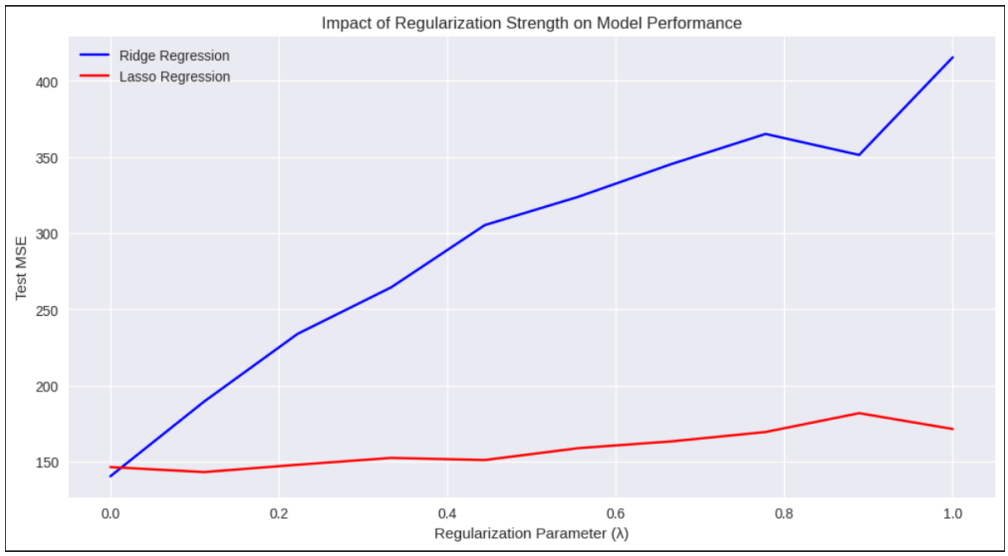

Figure 16: Impact of Regularisation

1.3.1 Observations

- **Ridge vs. Lasso at $\lambda = 0.5$:** Lasso regression outperforms Ridge, achieving a significantly lower Test MSE (155.37 vs. 307.11) and a higher R^2 score (0.76 vs. 0.54), indicating that Lasso provides better generalization for this dataset.
- **Effect of Increasing λ on Ridge:** As λ increases from 0 to 1, Ridge Regression Test MSE steadily rises (137.43 $\rightarrow$ 387.76), and R^2 decreases (0.765 $\rightarrow$ 0.397), showing that excessive regularization harms model performance.
- **Effect of Increasing λ on Lasso:** Lasso is more stable than Ridge, but beyond $\lambda = 0.33$, Test MSE starts increasing (137.48 $\rightarrow$ 169.94) while R^2 drops (0.765 $\rightarrow$ 0.7527), suggesting that overly strong L1 regularization leads to underfitting.
- **Best λ for Ridge and Lasso:** Both models achieve their best performance near $\lambda = 0$, where they behave like standard regression models with minimal penalty, reinforcing that lower regularization is preferable for this dataset.

2 Gradient Descent Algorithms: Comparison and Analysis

2.1 Types of Gradient Descent Algorithms

Gradient Descent is an optimization algorithm used for training machine learning models. The three primary types are:

2.1.1 Batch Gradient Descent

Description: Computes the gradient using the entire dataset before updating parameters.

Advantages:

- Stable convergence as it follows the true gradient direction.
- Works well for convex and smooth functions.

Disadvantages:

- Computationally expensive for large datasets.
- Can be slow to converge.

2.1.2 Stochastic Gradient Descent (SGD)

Description: Updates parameters using a single randomly chosen sample at each iteration.

Advantages:

- Faster updates, making it suitable for large datasets.
- Can escape local minima due to randomness.

Disadvantages:

- High variance in updates leads to oscillations in convergence.
- May require more iterations.

2.1.3 Mini-Batch Gradient Descent

Description: Uses small random subsets (mini-batches) of data to compute gradients.

Advantages:

- Faster than Batch Gradient Descent, more stable than SGD.
- Reduces variance in updates, leading to smoother convergence.

Disadvantages:

- Requires careful selection of batch size.
- May still oscillate in the presence of noisy gradients.

2.2 Convergence Analysis of Gradient Descent Methods

Fastest Convergence: Mini-Batch Gradient Descent converged the fastest, balancing stability and efficiency. SGD was highly oscillatory, while Batch Gradient Descent was slow.

2.3 Effect of Lasso and Ridge Regularization

2.3.1 Influence of Regularization

- **Lasso Regression:** Shrinks some feature weights to zero, performing feature selection.
- **Ridge Regression:** Reduces weight magnitudes but retains all features.

2.3.2 Optimal Lambda Values

Based on test performance:

- **Optimal λ for Ridge:** 0.0 (performed best with minimal regularization).
- **Optimal λ for Lasso:** 0.22 (best balance of feature selection and test performance).

2.4 Impact of Feature Scaling

Feature scaling improves gradient-based models by:

- Ensuring uniformity in gradient updates.
- Improving convergence speed.
- Enhancing regularization effectiveness.

2.5 Barplot of Trained Model Weights

Figure 17 compares feature weights for the best-performing models of Lasso and Ridge, as well as a non-regularized model.

2.6 Feature Influence on Delivery Time

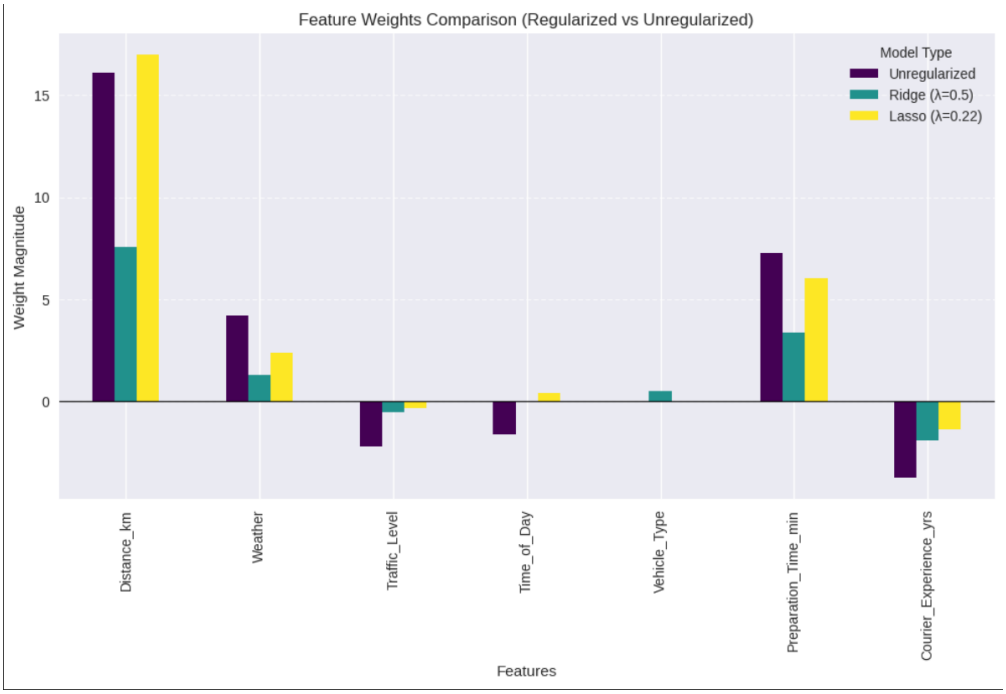

Figure 17: Comparison of Feature Weights (Regularized vs. Non-Regularized)

2.7 Analysis of Feature Weights

- Features with **higher absolute weight values** have stronger impacts on delivery time like Preparation time and Distance in km.
- Lasso identified **irrelevant features** by setting weights to nearly zero like Time of day and Vehicle Type.
- Ridge reduced influence but retained all features.

2.8 Features with Near-Zero Impact

- Lasso suggests some features have little to no effect on delivery time like traffic level, time of Day and Vehicle type as their weight values are very low.
- These features could be removed to reduce model complexity.

3 KNN and ANN Analysis

3.1 KNN Statistics

```
Summary Statistics:

Best performing configurations:
Best accuracy: 0.9207
k: 10
metric: euclidean

Accuracy by metric:
metric
cosine       0.914800
euclidean    0.914833
Name: accuracy, dtype: float64

Accuracy by k:
k
1     0.90480
5     0.91895
10    0.92070
Name: accuracy, dtype: float64
Best configuration: k=10, metric=euclidean
Best accuracy: 0.9207

Average accuracy by metric:
metric
cosine       0.914800
euclidean    0.914833
Name: accuracy, dtype: float64
```

Figure 18: Statistics

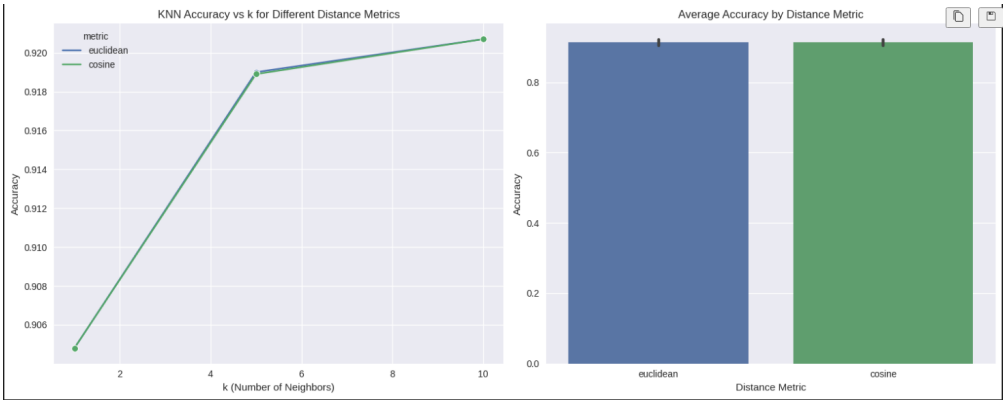

Figure 19: Plots

Text embedding classification accuracy:

```
Processing queries: 100%|██████████| 1000/1000 [00:00<00:00, 6770.50it/s]
Processing queries: 100%|██████████| 1000/1000 [00:00<00:00, 7116.47it/s]
Processing queries: 100%|██████████| 1000/1000 [00:00<00:00, 6610.89it/s]
Processing queries: 100%|██████████| 1000/1000 [00:00<00:00, 6215.06it/s]
Processing queries: 100%|██████████| 1000/1000 [00:00<00:00, 6737.13it/s]
Processing queries: 100%|██████████| 1000/1000 [00:00<00:00, 6676.48it/s]
Processing queries: 100%|██████████| 1000/1000 [00:00<00:00, 5945.02it/s]
Processing queries: 100%|██████████| 1000/1000 [00:00<00:00, 5675.17it/s]
Processing queries: 100%|██████████| 1000/1000 [00:00<00:00, 7021.34it/s]
Processing queries: 100%|██████████| 1000/1000 [00:00<00:00, 7208.23it/s]
Classifying: 100%|██████████| 10/10 [00:01<00:00,  6.37it/s]
Text embedding classification accuracy: 0.8781
```

Figure 20: Text-embedding classification

3.1.1 Retrieval Performance

Text to Image Retrieval Metrics:

- Mean Reciprocal Rank: 1.0000
- Precision@100: 0.9740
- Hit Rate: 1.0000

Image to Image Retrieval Metrics:

- Mean Reciprocal Rank: 0.9348
- Precision@100: 0.8411
- Hit Rate: 0.9996

3.2 Locally Sensitive Hashing

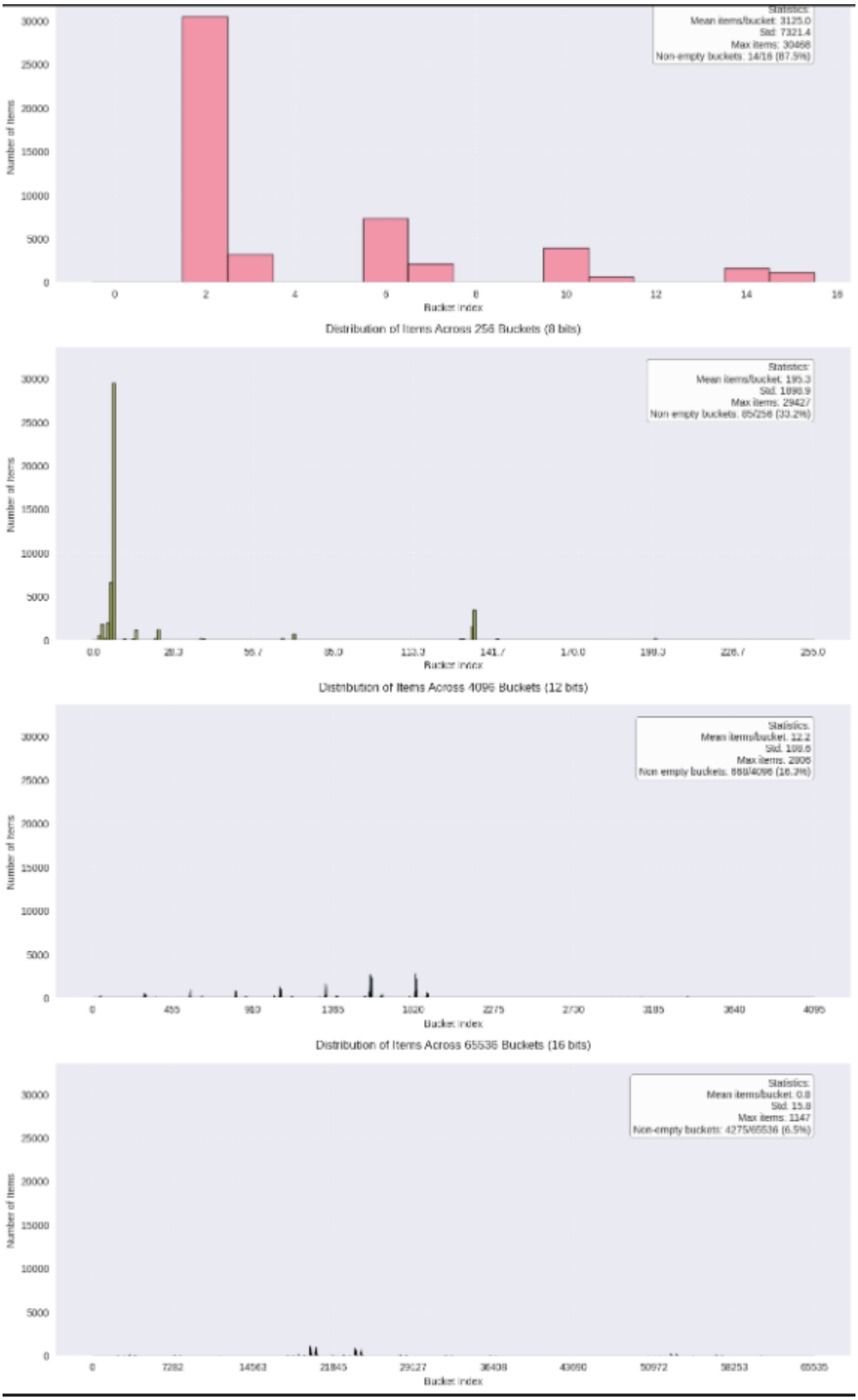

Figure 21: Histograms of bucket distribution

Problem :- As for higher value of nbits, most of the buckets remains empty and few of them are filled and also very sparsely.

```
Evaluating queries: 100%|██████████| 10000/10000 [03:30<00:00, 47.48it/s]

Results for 4 bits:
MRR: 0.9210
Precision@100: 0.7982
Hit Rate: 0.9991
Evaluating queries: 100%|██████████| 10000/10000 [01:17<00:00, 128.94it/s]

Results for 8 bits:
MRR: 0.9145
Precision@100: 0.7478
Hit Rate: 0.9976
Evaluating queries: 100%|██████████| 10000/10000 [00:55<00:00, 179.50it/s]

Results for 12 bits:
MRR: 0.9064
Precision@100: 0.6549
Hit Rate: 0.9919
Evaluating queries: 100%|██████████| 10000/10000 [00:11<00:00, 835.25it/s]

Results for 16 bits:
MRR: 0.8694
Precision@100: 0.4298
Hit Rate: 0.9662
```

Figure 22: Metrics of image to image retrival for k=100

4 Impact of Bit Count on Performance Metrics

4.0.1 Impact on Mean Reciprocal Rank (MRR)

- 4 bits: 0.9210
- 8 bits: 0.9145
- 12 bits: 0.9064
- 16 bits: 0.8694

We observe a gradual decrease in MRR as the number of bits increases. This decline occurs because:

- With more hyperplanes, the hash space becomes more fragmented, making it less likely for similar items to share the exact same hash code.
- Higher bit counts create more specific and smaller buckets, potentially separating vectors that are actually close neighbors.
- The first relevant match might end up in a different bucket, pushing it lower in the ranking and reducing MRR.

4.0.2 Impact on Precision@100

- 4 bits: 0.7982
- 8 bits: 0.7478
- 12 bits: 0.6549
- 16 bits: 0.4298

Precision shows the most dramatic decline as the number of bits increases. This happens because:

- More hyperplanes create a more fine-grained partition of the space, leading to smaller buckets.
- Similar items have a higher chance of being split across different buckets.
- With 16 bits, there are $2^{16} = 65,536$ possible buckets, which severely fragments the space.
- The dramatic drop from 0.7982 to 0.4298 suggests that many true similar pairs are being separated into different buckets at higher bit counts.

4.0.3 Impact on Hit Rate

- 4 bits: 0.9991
- 8 bits: 0.9976
- 12 bits: 0.9919
- 16 bits: 0.9662

Hit Rate shows the smallest decline, remaining relatively high even with more bits because:

- Even with high fragmentation, there's still a good chance of finding at least one relevant item.
- The binary nature of hit rate (finding at least one match) makes it more robust to increased partitioning.
- The high hit rates across all bit counts suggest the LSH scheme is still maintaining some effectiveness at preserving similarity relationships.

4.0.4 Key Insights and Trade-offs

4.0.5 Speed vs. Accuracy Trade-off

- More bits (16) result in faster query times (835.25 it/s) compared to fewer bits (4 bits: 47.48 it/s).
- However, this comes at the cost of reduced accuracy across all metrics.

4.0.6 Bucket Size Trade-off

- Fewer bits (4) create larger buckets, capturing more similar items but requiring more distance computations.
- More bits (16) create smaller, more specific buckets, leading to faster queries but potentially missing similar items.

4.0.7 Optimal Balance

- 8 bits shows a good balance with reasonable performance (128.94 it/s) while maintaining strong metrics.
- The 12-bit configuration might be preferred if query speed is more important than precision.
- 16 bits shows too much degradation in precision to be practical unless speed is the absolute priority.

4.0.8 Practical Implications

- For applications requiring high precision, using fewer bits (4-8) is recommended.
- For large-scale applications where query speed is crucial and some accuracy can be sacrificed, higher bit counts could be considered.
- The choice of bit count should be based on the specific requirements of precision vs. speed in the application.

4.1 IVF Implementation

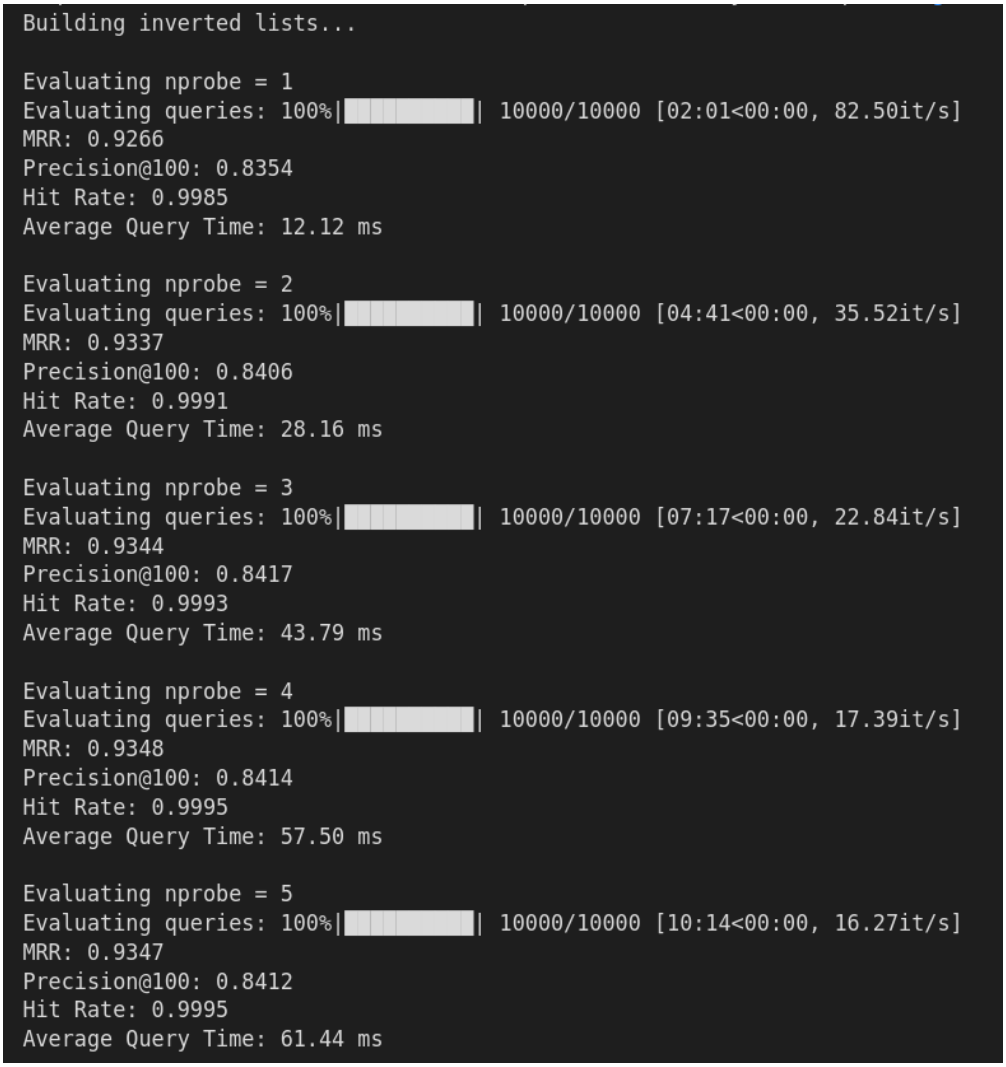

Figure 23: Image to image retrieval metrics

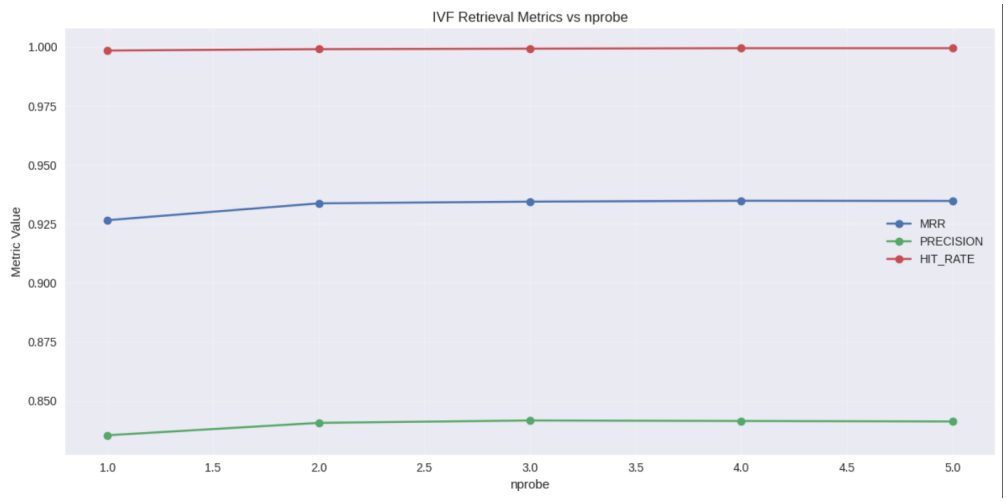

Figure 24: Metrics plot

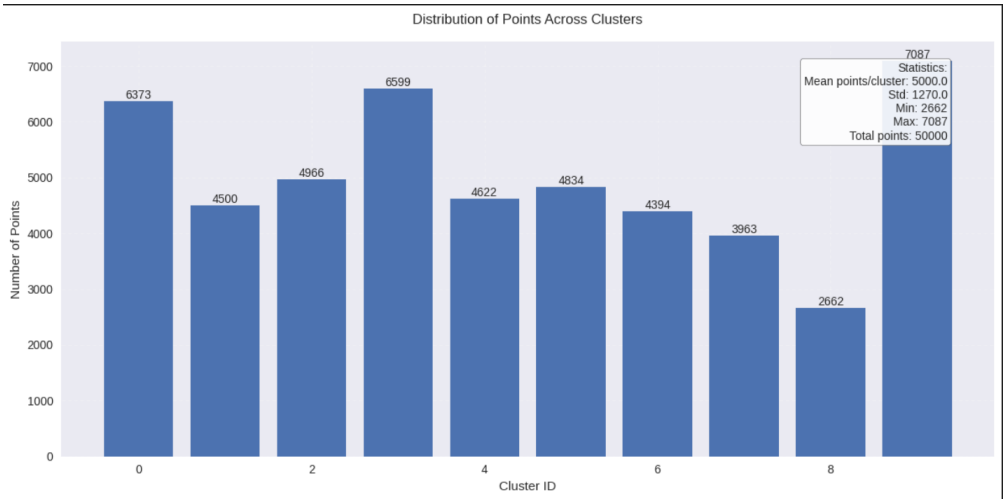

Figure 25: Points accross clusters

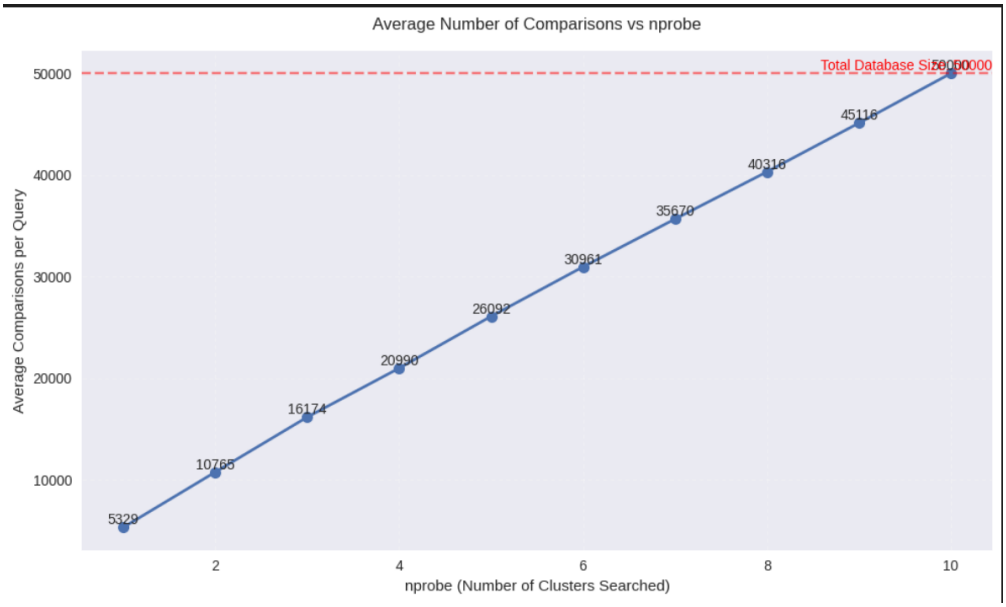

Figure 26: Average number of comparisons vs nprobe

Analysis between all three searching techniques:-

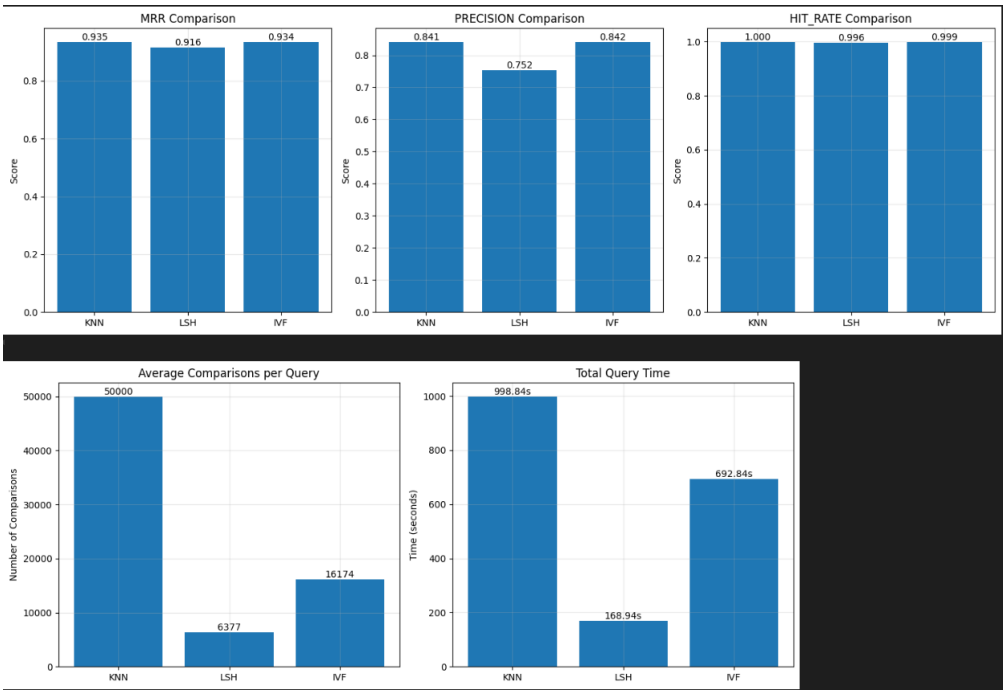

Figure 27: Comparison between 3 meth

5 The Crypto Detective

5.1 Data Analysis and Preprocessing

[Insert correlation matrix heatmap] [Insert violin plots for feature distributions]

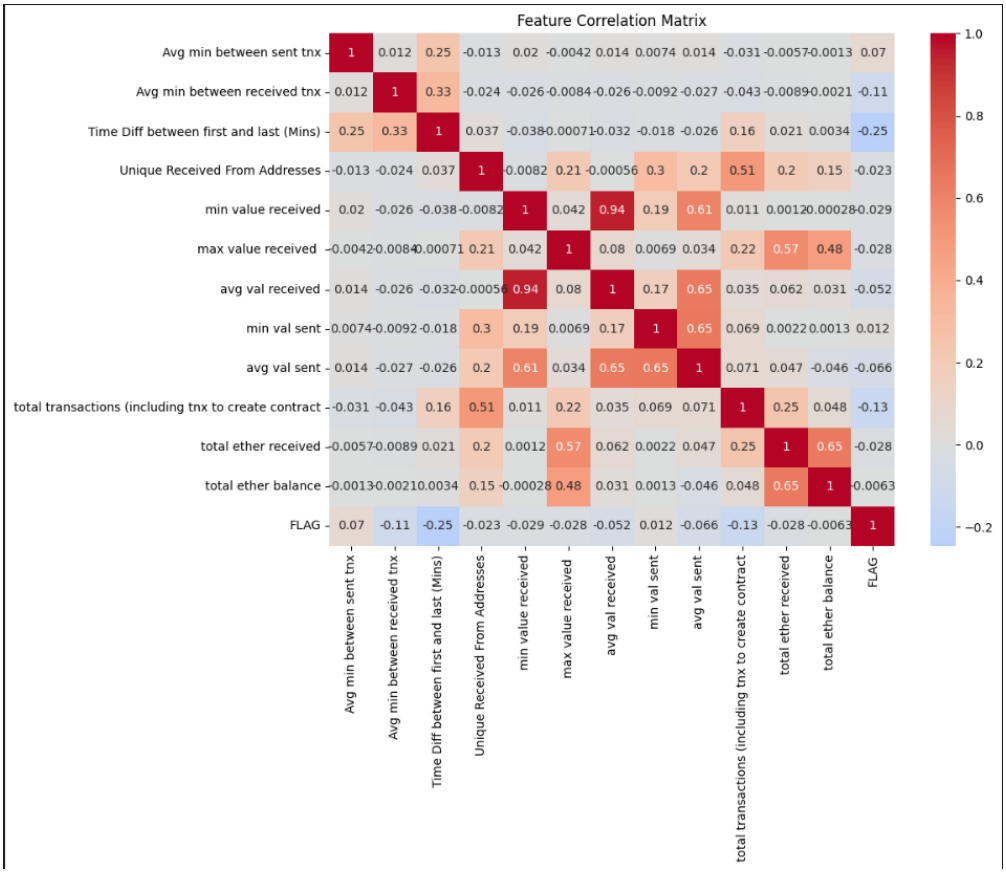

Figure 28: Correlation heatmap

Observations from the Correlation Matrix

Strong Positive Correlations:

- **avg val received** has a very strong positive correlation with **min value received** (**0.94**). This indicates that as the average value of received transactions increases, the minimum value received also tends to increase proportionally.
- **avg val sent** is highly correlated with **avg value received** and **min val sent**.
- **total ether received** has a strong positive correlation with **total ether balance** (**0.65**), showing that accounts with higher ether received tend to retain more ether in their balance.

Moderate Positive Correlations:

- **Unique Received From Addresses** is moderately correlated with **total transactions (including trx to create contract)** (**0.51**). This implies that accounts interacting with more unique addresses are involved in more transactions.
- **max value received** shows moderate correlation with **total ether received** (**0.57**) and **total ether balance** (**0.48**). Higher maximum transaction values received contribute to both ether received and the balance.

Negative Correlations with the Target Variable (FLAG):

- **Avg min between received tnx** has a negative correlation with **FLAG** (**-0.11**). This suggests that accounts with longer intervals between receiving transactions may have a lower likelihood of the flag being triggered.
- **Time Diff between first and last (Mins)** is negatively correlated with **FLAG** (**-0.25**). Accounts active over a longer time tend to be less likely flagged.

Weaker or No Correlation:

- Features like **avg val sent, min val sent**, and **Unique Received From Addresses** have almost no significant correlation with **FLAG**, indicating that these features might not be as relevant for predicting the target variable.
- **total transactions (including trx to create contract)** shows weak negative correlation with **FLAG** (**-0.13**).

Interesting Observations in Feature Relationships:

- **min val sent** and **avg val sent** are highly correlated (**0.65**), suggesting a proportional relationship between the minimum and average values sent in transactions.
- **total ether received** has strong correlations with both **total ether balance** and **avg val received**, indicating that ether balance and average received value are influenced by the overall ether received.

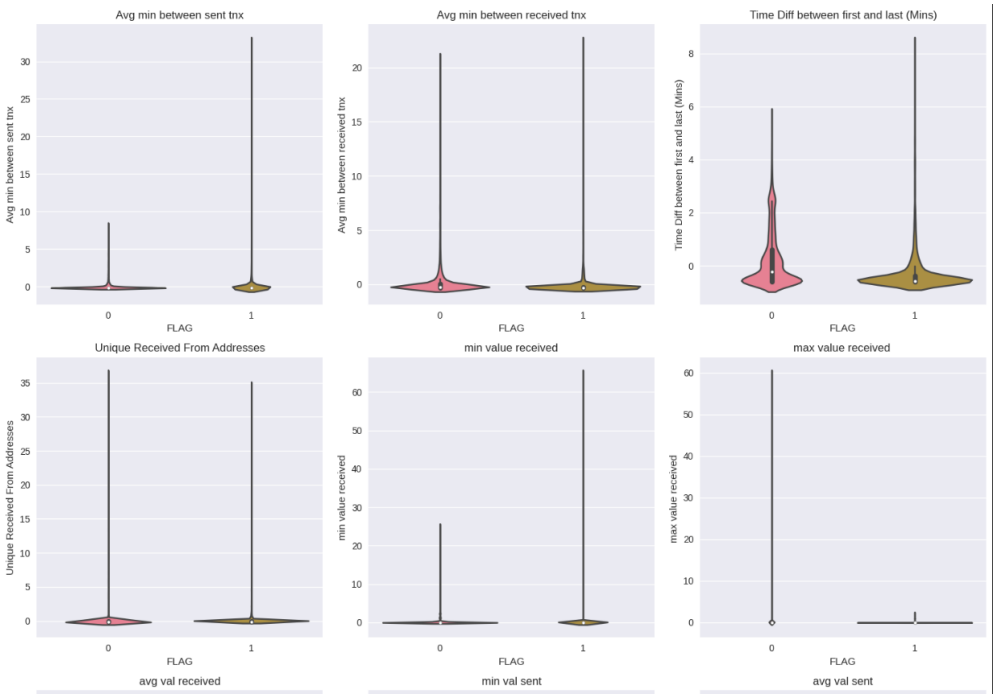

Figure 29: Violin plots

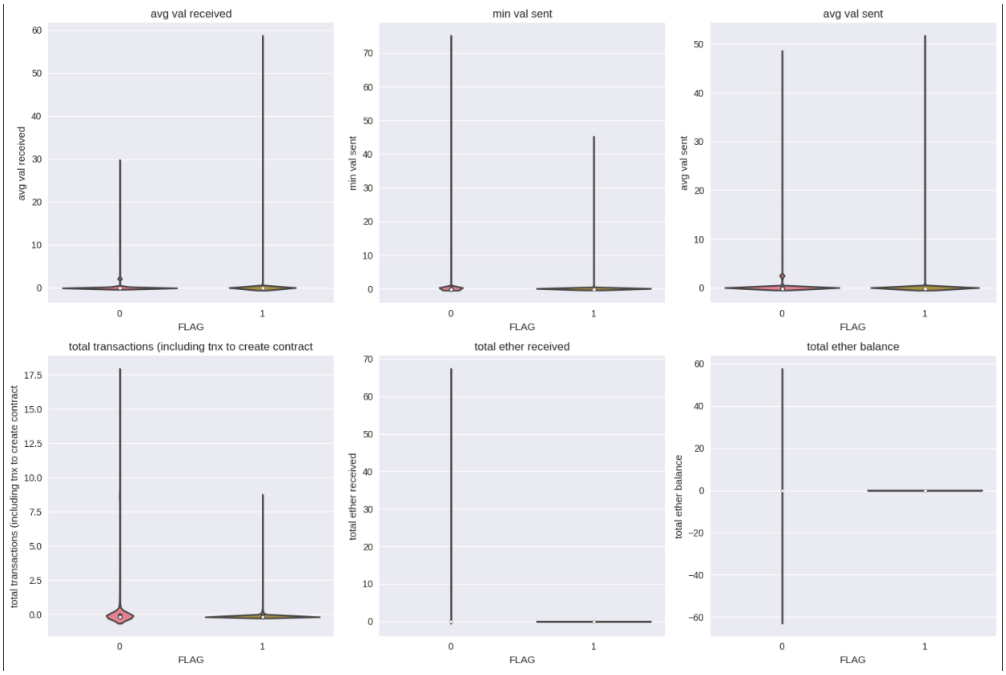

Figure 30: Violin plots

Observations from the Violin Plots

Avg min between sent tnx:

- For both FLAG = 0 and FLAG = 1, the distribution is heavily concentrated near zero.
- A small number of outliers exist, with slightly longer intervals for FLAG = 0.

Avg min between received tnx:

- The distribution for FLAG = 1 is more spread out compared to FLAG = 0.
- FLAG = 0 has a significant concentration near zero, while FLAG = 1 shows a wider range of intervals.

Time Diff between first and last (Mins):

- The distribution for FLAG = 1 is broader, indicating that flagged accounts are active over shorter periods.
- For FLAG = 0, accounts have smaller and longer time differences, with fewer accounts having long periods of activity.

Unique Received From Addresses:

- The distribution for both FLAG = 0 and FLAG = 1 is concentrated at the lower end, suggesting most accounts interact with only a few unique addresses.
- A small subset of flagged accounts interacts with many addresses.

min value received:

- Both FLAG = 0 and FLAG = 1 have distributions concentrated near the lower values, with a few flagged accounts receiving higher minimum values.

max value received:

- Flagged accounts (FLAG = 1) show a broader spread in the maximum value received, with some accounts receiving significantly higher values compared to unflagged accounts (FLAG = 0).

avg val received:

- The average value received is concentrated near lower values for both flags, but flagged accounts exhibit a wider range.

min val sent:

- Both FLAG = 0 and FLAG = 1 show similar distributions, with most transactions concentrated at lower minimum values sent.

avg val sent:

- Flagged accounts (FLAG = 1) have a broader spread, indicating higher variability in the average value sent.

total transactions (including trx to create contract):

- Accounts with FLAG = 1 have a slightly broader spread in the total number of transactions, suggesting that flagged accounts may engage in a wider range of activities.

total ether received:

- The distribution for both flags is concentrated near zero, with very few accounts receiving significant amounts of ether.
- Flagged accounts (FLAG = 1) show slightly higher values in rare cases.

total ether balance:

- Flagged accounts have a broader range of balances, with some showing negative balances, while unflagged accounts (FLAG = 0) are primarily concentrated near zero.

General Observations

- Accounts with FLAG = 1 (flagged) generally exhibit broader distributions across most features, indicating higher variability in behavior.
- Unflagged accounts (FLAG = 0) tend to have more concentrated distributions, with lower ranges for most features.
- Features like **Time Diff between first and last (Mins)** and **max value received** show distinct differences between FLAG = 0 and FLAG = 1, which could be useful for distinguishing flagged accounts.

5.2 Decision Tree Implementation

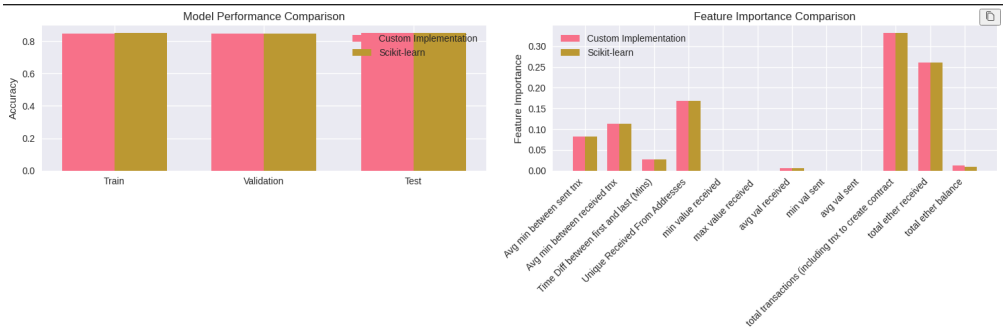

Figure 31: Model Comparison

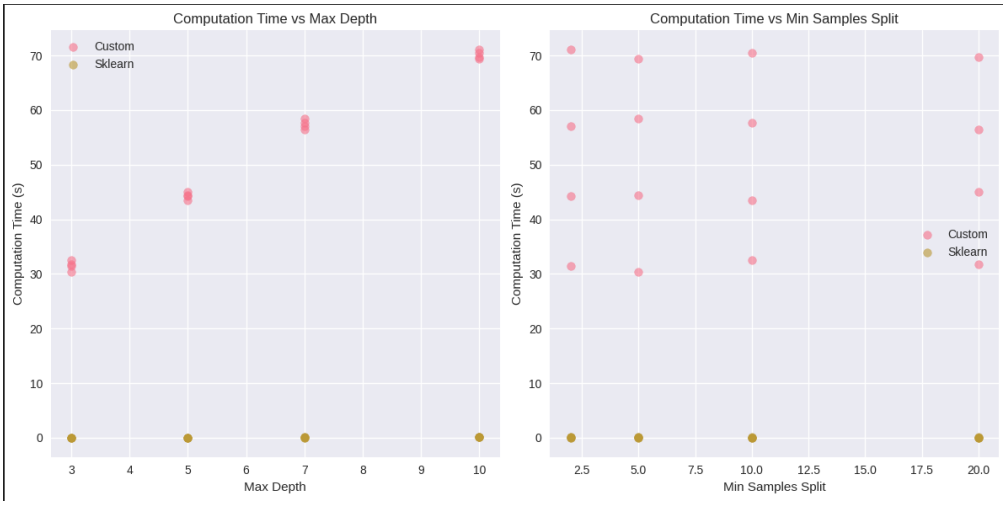

Figure 32: Computation Time comparison

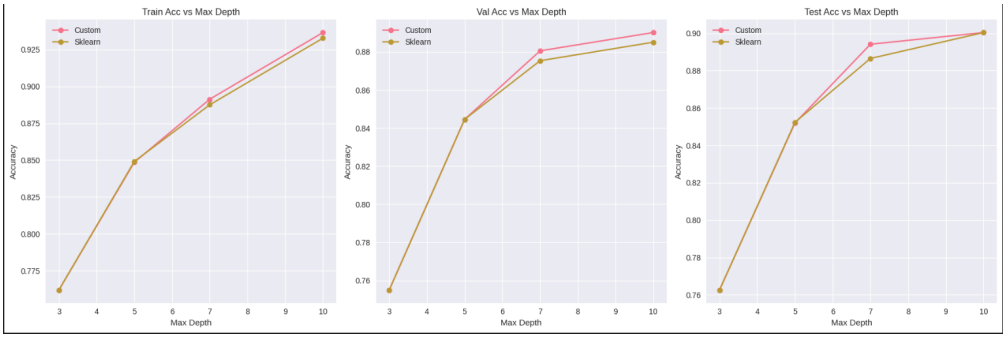

Figure 33: Accuracy with max depth

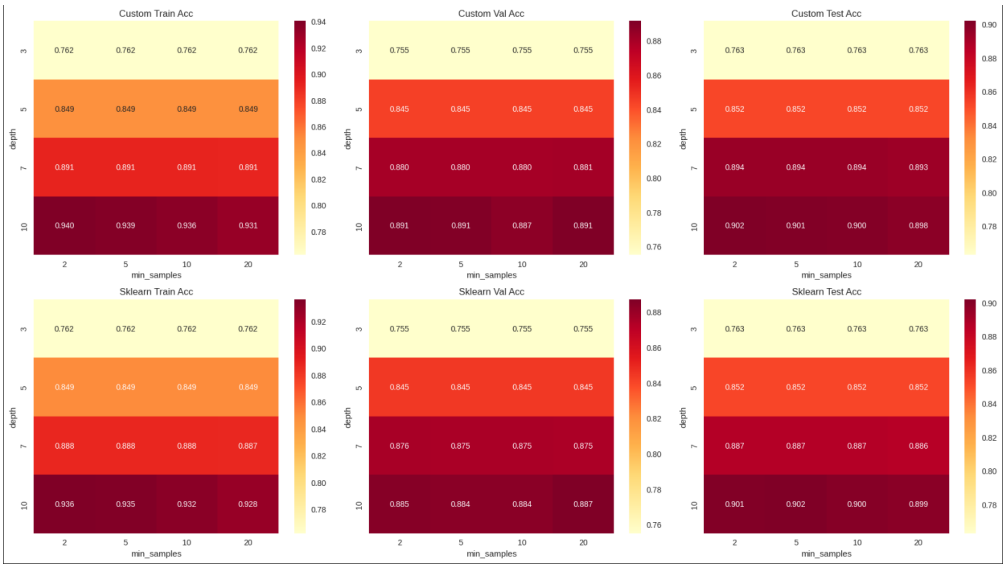

Figure 34: Heatmap

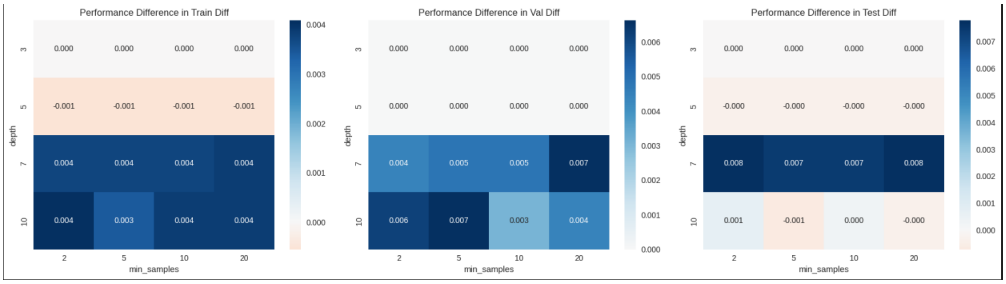

Figure 35: Heatmap

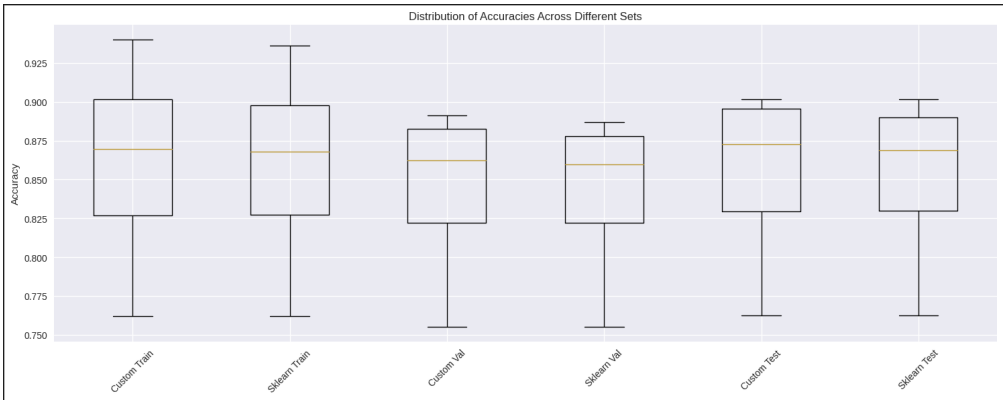

Figure 36: Box Plot

```
Performance Summary:

Accuracy Statistics:

Train Acc:
Custom Implementation:
  Mean: 0.8596
  Std: 0.0665
  Max: 0.9402

Sklearn Implementation:
  Mean: 0.8578
  Std: 0.0649
  Max: 0.9361

Val Acc:
Custom Implementation:
  Mean: 0.8426
  Std: 0.0551
  Max: 0.8914

Sklearn Implementation:
  Mean: 0.8400
  Std: 0.0530
  Max: 0.8870
...

Sklearn Implementation:
  Mean: 0.0444 seconds
  Std: 0.0169 seconds
```

Figure 37: Summary

6 Observations

6.1 Accuracy Comparison

- The accuracies and feature importance of both custom and in-built scikit-learn decision tree implementations are nearly the same.
- After `max_depth` > 5, the training, testing, and validation accuracies of the custom implementation are slightly higher than those of scikit-learn.
- **Accuracy Details:**
  - **Training:** The custom model achieves the highest accuracy of 0.94, while scikit-learn

reaches 0.936 at `max_depth` = 10 and `min_samples_split` = 2.

- **Testing:** The highest accuracy for both models is 0.902 at `max_depth` = 10.
- **Validation:** The custom model reaches the highest accuracy of 0.91, whereas scikit-learn achieves 0.887.

6.2 Computation Time

- Increasing `max_depth` and `min_samples_split` leads to significantly faster computation times for the scikit-learn model.
- The scikit-learn implementation takes approximately 0.044 seconds, whereas the custom implementation takes around 30 to 70 seconds.

6.3 Analysis

The significant difference in computation time between the custom decision tree implementation (30-70 seconds) and scikit-learn's `DecisionTreeClassifier` (0.044 seconds) can be attributed to several key factors:

6.4 Optimized Implementation in Scikit-learn

- Scikit-learn's decision tree is implemented in Cython, a compiled language that is much faster than Python.
- It utilizes efficient memory management and highly optimized data structures (such as NumPy arrays) for fast traversal and computations.

6.5 Algorithmic Optimizations

- Scikit-learn leverages vectorized operations, reducing the need for Python loops, which are inherently slow.
- It employs efficient splitting heuristics, including best split selection using sorted values and precomputed statistics.
- Pruning techniques and optimized node splitting strategies further improve speed.

6.6 Data Structure Differences

- The custom implementation may rely on lists and dictionaries, which are slower to traverse and modify than NumPy-based approaches.
- Scikit-learn likely uses efficient indexing techniques (e.g., binary search, heap structures) to minimize computational overhead.

6.7 Default Hyperparameters

- Scikit-learn's default hyperparameters (e.g., `min_samples_split`=2, `max_features`=None, `min_impurity_decrease`=0.0) ensure a balance between accuracy and speed.
- The custom implementation may lack optimizations such as early stopping or best split computation, leading to exhaustive tree growth.

6.8 Parallelization and Low-Level Optimizations

- Scikit-learn utilizes parallel processing, especially for tree-based models like RandomForest.
- It avoids redundant calculations by caching results and reusing computed values.
- The custom implementation is likely single-threaded and may perform redundant calculations at each node.

6.9 Python Overhead

- Python, being an interpreted language, runs significantly slower than compiled Cython or C++ implementations in scikit-learn.
- Function calls, recursion, and memory allocation in Python introduce additional computational overhead.

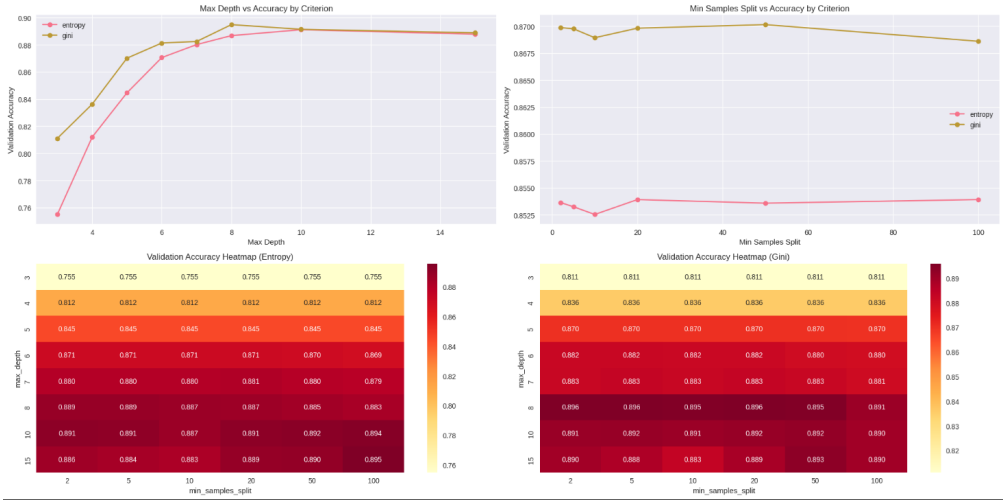

Figure 38: Accuracy vs criterion

```
Hyperparameter Tuning Results:

Best Configuration:
Criterion: gini
Max Depth: 8
Min Samples Split: 2
Validation Accuracy: 0.8958
Training Accuracy: 0.9277

Analysis:

1. Impact of Max Depth:
max_depth
3     0.783002
4     0.824062
5     0.857358
6     0.875938
7     0.881273
8     0.890802
10    0.891244
15    0.888300
Name: val_accuracy, dtype: float64

2. Impact of Min Samples Split:
...

4. Overfitting Analysis:
Mean overfitting gap: 0.0209
Max overfitting gap: 0.1030
```

Figure 39: Hyperparameter tuning

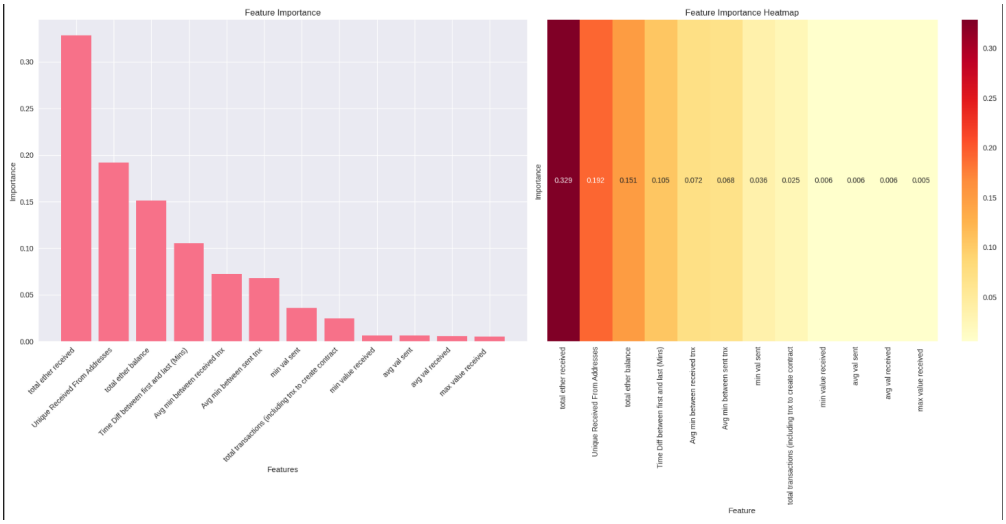

Figure 40: Feature Importance

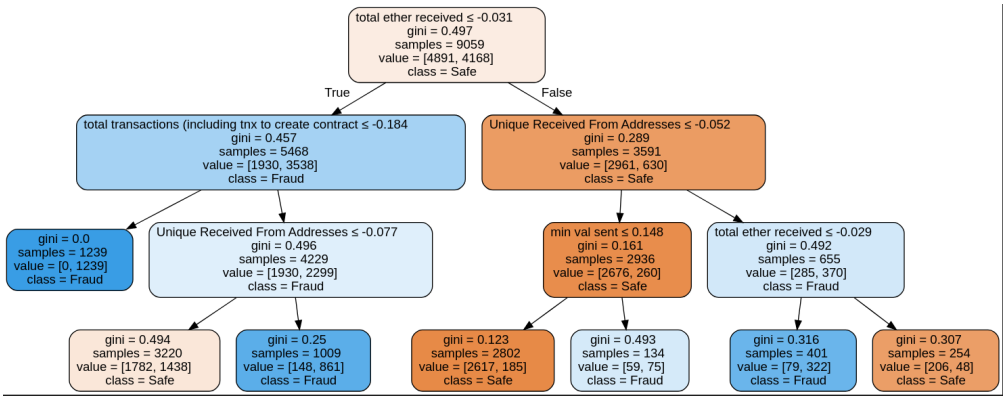

Figure 41: Tree visualisation

7 K-Means and SLIC Implementation

7.1 SLIC Algorithm Results

Figure 42: Original Image

Figure 43: Image with m=10 and k=200

Figure 44: Image with m=20 and k=200

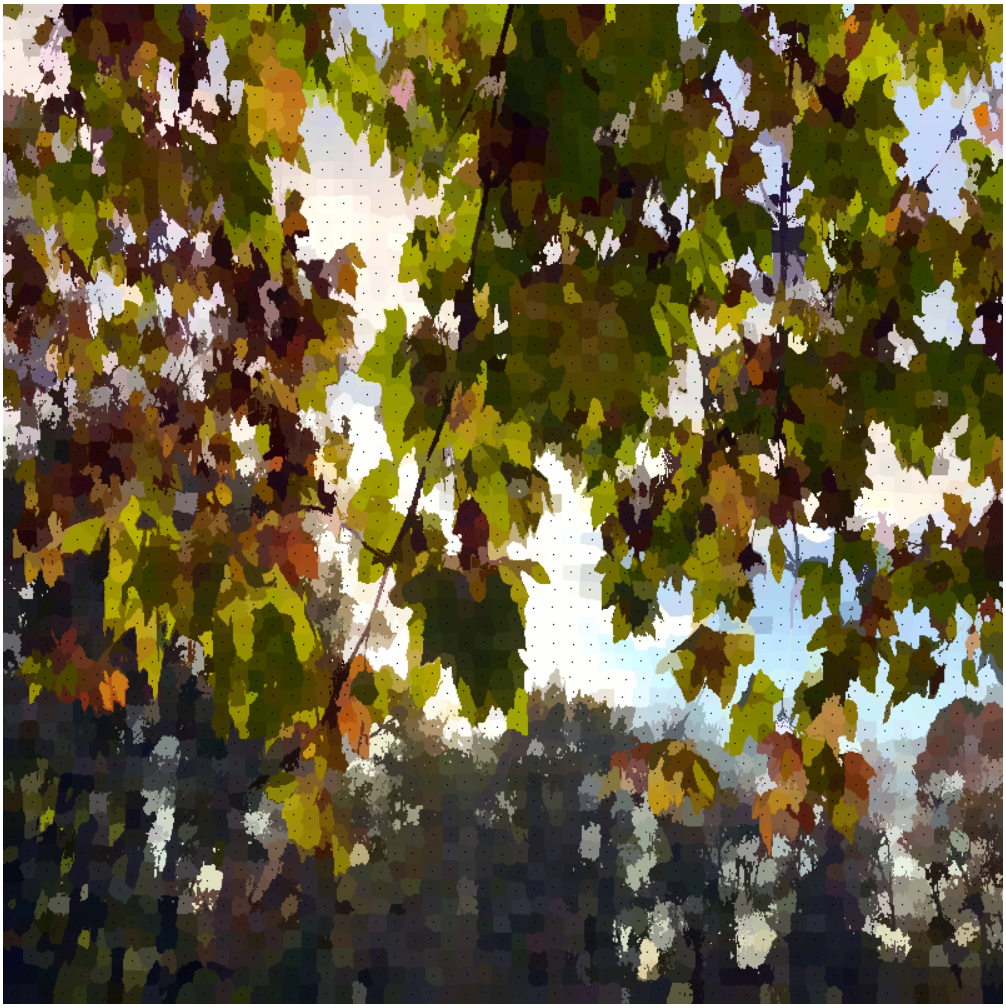

Figure 45: Image with m=20 and k=3000

So can be seen with with high m , image segmentation is better and more clearly clusters can be seen
Also with high value of k small-sized clusters will be made

Analysis of parameters:

- Impact of number of clusters - Higher K leads to computational costs and smaller sized superpixels
- Effect of compactness parameter - Higher m produces more regularly shaped superpixels while lower m allows superpixels to adapt better to edges and textures.
- RGB vs Lab color space comparison - RGB directly represents image colors, preserving them as they appear while LAB is perceptually uniform but not idea for direct segmentation. RGB maintains smooth color blending while LAB distort relationships.

7.2 Video Segmentation

Video of segmenting frames and video creation without optimisation

7.3 Optimisation

Video of segmenting frames and video creation with optimisation

Key Optimizations

1. Uses previous frame's cluster centers as starting points for the next frame.
2. Processes frames in local regions (ROIs) instead of the entire image for each cluster.
3. Implements vectorized operations with `numpy` for faster distance calculations.
4. Employs early stopping when changes between iterations are below tolerance.
5. Restricts search space to local neighborhoods around cluster centers.
6. Uses a moderate number of clusters (50) balanced for speed and quality.
7. Sets a practical tolerance threshold (500) for convergence checking.
8. Keeps track of iterations per frame to measure optimization effectiveness.
9. Applies carefully tuned compactness factor (10) for balance between speed and accuracy.
10. Maintains frame-to-frame temporal consistency through cluster center inheritance.

Results

The number of iterations per frame reduces from 10 to 3.64 after optimization, achieving a significant improvement while maintaining segmentation quality.

$$\text{Iterations per frame:} \quad 10 \rightarrow 3.64$$