STA 160

Final Project

Tracing the Ink: What Makes a Digit Unique?

Jayden Kwong

1. Introduction

In 1999, the National Institute of Standards and Technology (NIST) collected a dataset of handwritten digits (0 to 9) from many individuals. This dataset, known as MNIST, has become the benchmark for evaluating image-based classification algorithms in machine learning. The dataset includes thousands of 28×28 grayscale images, where each image is a hand-drawn digit with pixel values indicating lightness or darkness.

The real-world relevance of handwritten digit recognition goes beyond academic exercises. I work as an intern at CalSTRS (California State Teachers' Retirement System), one of the largest public pension funds in the United States, managing retirement benefits for California's educators. As part of its operations, CalSTRS frequently processes handwritten checks from retirees and beneficiaries to manage pension fund disbursements and contributions. Every day, thousands of checks flow through automated systems to ensure accurate processing and secure fund management. My experience seeing these automated check readers in action has piqued my curiosity about how these systems reliably interpret handwritten digits, despite natural variations in handwriting styles. Understanding how to classify these handwritten digits connects directly to my daily work at CalSTRS and highlights how computer vision and pattern recognition are crucial in modern financial data processing.

My research question is: **How does the ink usage of handwritten digits vary across different digits, and what can these variations reveal about their visual complexity for classification?** To address this, I explore how engineered features can reveal the visual complexity of handwritten digits. Using these features, I will build and compare classification models (Random Forest, Gradient Boosting, and KNN) to see how well they can distinguish digits based solely on these interpretable features. I will also conduct hyperparameter tuning to optimize model performance and understand how these simple features can capture the complexity of handwritten digit shapes.

Ultimately, this project aims to demonstrate that even with simple, engineered features, it is possible to model and understand the intricate patterns of handwritten digits — patterns that are central to automated systems in the real world.

2. Exploratory Data Analysis (EDA)

The dataset consists of 42,000 grayscale images of handwritten digits (0–9). Each image has 784 pixel columns with values from 0 (white) to 255 (black), plus a digit label. The dataset is already cleaned, with no missing values.

To better understand and reduce the dimensionality of this dataset, we engineered six new features that capture the overall "ink usage" and variability of each digit:

1. Total Ink: Sum of pixel intensities per image.
2. Average Intensity: Mean pixel intensity.
3. Standard Deviation: Standard deviation of pixel intensities.
4. Active Pixels: Count of nonzero pixels (measures how many pixels were used).
5. Minimum Nonzero Intensity: Lightest active pixel.

6. Maximum Intensity: Darkest pixel.

Including the digit label, we now have a dataset with seven columns and 42,000 data points.

2.1 Digit Count Distribution
Figure 1 shows the number of samples for each digit. The dataset is well-balanced across all digits, with around 4,000–4,500 samples per digit. This balance ensures our models will not unfairly favor any specific digit.

2.2 Total Ink Usage by Digit
Figure 2 presents boxplots of total ink usage for each digit. Digits with more curves and strokes (like 0 and 8) have higher median ink usage, while simpler digits (like 1) have lower usage. This suggests that total ink is a meaningful feature to help models differentiate digits.

2.3 Feature Correlations
Figure 3 shows the pairwise correlations between the engineered features. Total_ink, avg_intensity, and std_intensity are extremely highly correlated ($r \approx 0.98$–1), suggesting they measure similar overall darkness and variability. Active_pixels is also strongly correlated ($r \approx 0.93$–0.96) with these features. In contrast, min_nonzero_intensity and max_intensity show very low correlation with the other features ($r \approx 0.01$–0.09). These capture extremes in pixel intensity, not overall ink usage.

2.4 Pixel Activity Heatmap
Figure 4 displays a pixel activity heatmap that highlights a bright central core—where most strokes appear in digit drawings. This explains why features like active pixel count and total ink intensity can help models distinguish digits: they capture how much of this core region is filled. Sparse outer areas confirm that digits are generally well-centered, minimizing noise.

2.5 Ink Usage vs. Active Pixels
Figure 5 illustrates the strong linear relationship between total ink usage and active pixel count across all digits. Simpler digits like 1 appear in the bottom left with low ink and pixel counts, while more complex digits like 8 and 0 appear in the top right with higher usage. This confirms that these two features both capture the complexity and fullness of digit strokes.

2.6 t-SNE Visualization
Figure 6 shows a two-dimensional t-SNE projection of the engineered features. Each point represents an individual digit, color-coded by label. Digits like 1 form compact clusters due to their simple, consistent shape, while digits like 8 are more diffuse, reflecting variation in how people draw them. Digits with similar shapes (like 3 and 5, or 8 and 9) cluster closer together or overlap, hinting at areas where classification may be more challenging. This plot confirms that our engineered features provide good separation of digits in feature space.

2.7 Conclusion of EDA
The engineered features show clear differences between digits and provide strong signals for downstream classification models. Next, we will explore feature importance and build classification models to test how well these features can distinguish handwritten digits.

3. Methodology
3.1 Feature Engineering and Importance
To capture the visual complexity of handwritten digits, we engineered features including total ink, average intensity, standard deviation of intensity, active pixel count, and extreme pixel values. To assess how much each feature contributes to classification, we used a Random Forest classifier to compute feature importance. We selected Random Forests because they naturally handle non-linear data and provide a direct measure of feature relevance by ranking feature importances. Understanding feature importance also helps inform our choice of features when training our final models.

3.2 Classification Models
Given the non-linear nature of handwritten digit data, we evaluated three models that can handle non-linear decision boundaries:

Random Forest: An ensemble of decision trees, each trained on different random subsets of the data. Random Forests help reduce variance and overfitting while capturing complex patterns.

Gradient Boosting: Similar to Random Forests in using decision trees, but trains them sequentially, with each tree learning to correct the errors of the previous ones. This boosting approach often improves predictive accuracy.

K-Nearest Neighbors (KNN): Classifies new samples by examining the K closest data points in the training set. This model is particularly useful for exploring local decision boundaries and assessing how well simple distance-based approaches can work with our engineered features. For KNN, we used cross-validation (CV=5) to select the optimal K.

We chose these models because of their balance of interpretability and ability to model non-linear data.

3.3 Data Splitting and Model Training
The dataset was split into 80% training and 20% testing to evaluate model generalization. We initially trained the models using all engineered predictors as a baseline, including tests with single predictors to assess their individual predictive power.

3.4 Evaluation Metrics
We used two categories of evaluation metrics:

1. Individual Class Metrics:
    a. Precision: Measures how many predictions for a specific digit are correct.
    b. Recall: Measures how well the model finds all instances of that digit.
    c. F1-Score: The harmonic mean of precision and recall, balancing the two.
    d. Support: The number of true samples for each digit, used to identify potential class imbalances that could bias the model.

2. Overall Model Metrics:
    a. Accuracy: The proportion of all predictions that are correct across the entire test set.
    b. Macro Average: The average of precision, recall, and F1-score across all classes, treating each class equally.
    c. Weighted Average: Similar to the macro average but weighted by the number of true samples for each class, reflecting the impact of class imbalance on overall performance.

These metrics provide complementary views of model performance: overall accuracy as a high-level measure, and class-specific metrics to evaluate how well each digit is recognized.

3.5 Modeling Approach

Our initial analysis involved training models using all engineered features to establish a baseline for predictive accuracy. We then evaluated each model's performance using the metrics described above. Hyperparameter tuning was conducted to optimize model performance, and results are discussed in the following section.

4. Results

4.1 Baseline Model Performance

We first established baseline performances for each classification model using all six engineered features. The accuracies on the test set were:

Model	**Accuracy**
Random Forest	27.4%
Gradient Boosting	30.2%
K-Nearest Neighbors	22.8%

These results suggest that Gradient Boosting initially achieved the highest accuracy, followed by Random Forest and KNN.

4.2 Feature Importance and Selection

To identify the most influential predictors, we examined the feature importance rankings (Figures 7 and 8).

Random Forest
Based on feature importance, Max Intensity and Minimum Nonzero Intensity were the least informative. Removing these two features reduced the test accuracy slightly to 27.2%, a decrease of only 0.2%.

Gradient Boosting
For Gradient Boosting, the recommended removal included Max Intensity, Total Ink, Average Intensity, and Minimum Nonzero Intensity. This reduced the accuracy to 29%, a decrease of 1.2%.

While these adjustments led to minor reductions in accuracy, we opted to follow the feature importance recommendations. Reducing the number of predictors enhances model interpretability and avoids potential redundancy in features.

Figure 9 presents the classification report for the baseline Random Forest model, highlighting performance across individual digits. Figure 10 shows the corresponding classification report for the final models after hyperparameter tuning.

4.3 Hyperparameter Tuning
To further optimize performance, we performed hyperparameter tuning (Figure 11), focusing on the number of trees (n_estimators) for Random Forest and Gradient Boosting, and the optimal number of neighbors (K) for KNN.

Random Forest: Best n_estimators = 200 (minimum test error: 72.19%)
Gradient Boosting: Best n_estimators = 140 (minimum test error: 69.49%)
KNN: Best K = 12 (minimum cross-validated test error: 23.93%)

4.4 Final Model Performance
After applying hyperparameter tuning with the reduced feature sets:

Random Forest (200 trees): Accuracy = 27.3%
Gradient Boosting (140 trees): Accuracy = 28.9%
KNN (K=12): Accuracy = 23%

Interestingly, for Gradient Boosting, the tuned model did not outperform the baseline model with default n_estimators=100. Therefore, we chose to retain the baseline Gradient Boosting model (100 trees), which provided the highest accuracy of 29% among the models tested.

5. Discussion and Reflection
This project successfully explored the classification of handwritten digits by engineering simple and interpretable features. While the overall accuracy of the best model (Gradient Boosting) is modest compared to deep learning models that work directly with pixel data, this approach demonstrates that even basic features can capture key aspects of handwritten digit complexity.

A significant strength of this analysis is its focus on interpretability. By summarizing pixel-level information into intuitive features, we gained insights into how visual complexity varies across different digits. Features like total ink and active pixels directly connect to the visual characteristics of handwritten digits such as how curvy or dense they are, making the models more transparent.

However, several limitations are also clear. The engineered features, while helpful, do not fully capture the rich spatial structure of handwritten digits. As seen in the t-SNE plots, digits like 8 and 0 overlap considerably with others, suggesting that shape-based features (like stroke orientation or loops) might be missing from our feature set. Similarly, the overall accuracy indicates that spatial dependencies between pixels are not fully represented in our feature-based approach.

In terms of assumptions and model diagnostics, we assumed that the relationships between features and digit labels were strong enough for these models to learn effectively. Random Forest and Gradient Boosting handle non-linearities well, but KNN can struggle in high-dimensional spaces. Visualizing feature correlations (Figure 3) helped us identify redundancy, which we addressed by dropping highly correlated features. The minimal change in accuracy after dropping these features suggests that the models' performance is robust to feature reduction, an encouraging finding for generalizability.

A challenge that was discovered was balancing model complexity with interpretability. While adding more features might improve accuracy slightly, it also increases the risk of overfitting and reduces the clarity of the insights gained. We chose to prioritize interpretability, especially given the real-world relevance of these features in contexts like check processing.

Looking ahead, this analysis could be extended in several ways. One natural next step would be to explore convolutional neural networks or other deep learning approaches that operate directly on pixel data. These models could exploit the spatial relationships we know are important for digit shapes. Additionally, feature engineering could be expanded to include pixel location-based features, such as center of mass or pixel density in specific regions of the digit.

Overall, this project demonstrates that even simple features can reveal important aspects of handwritten digit complexity. It also highlights the trade-offs between interpretability and performance, and underscores the importance of carefully considering these trade-offs when choosing a modeling approach.

6. Conclusion

This project set out to explore how ink usage varies across handwritten digits and what these differences can reveal about their visual complexity. By engineering a set of interpretable features, we built and evaluated several models, ultimately achieving the best performance with a Gradient Boosting model that reached an accuracy of 29%.

While this accuracy is modest compared to more advanced image recognition techniques, it’s important to recognize what this number represents: these engineered features alone explain about 29% of the total variation in handwritten digit classification. In other words, simple measures like total ink usage and active pixel count capture nearly one-third of the visual complexity of handwritten digits, highlighting their value in understanding digit shape variation and analysis.

This analysis demonstrates that even straightforward, interpretable features can provide meaningful insights into handwritten digit recognition. Although the accuracy did not reach our initial expectations, the results show that ink complexity and related measures have a measurable influence on classification performance. This underscores the importance of combining intuitive features with more sophisticated modeling approaches in future work to further close the gap in predictive performance.

Appendix

Reference

Dataset: https://www.kaggle.com/competitions/digit-recognizer/data

Figure 1

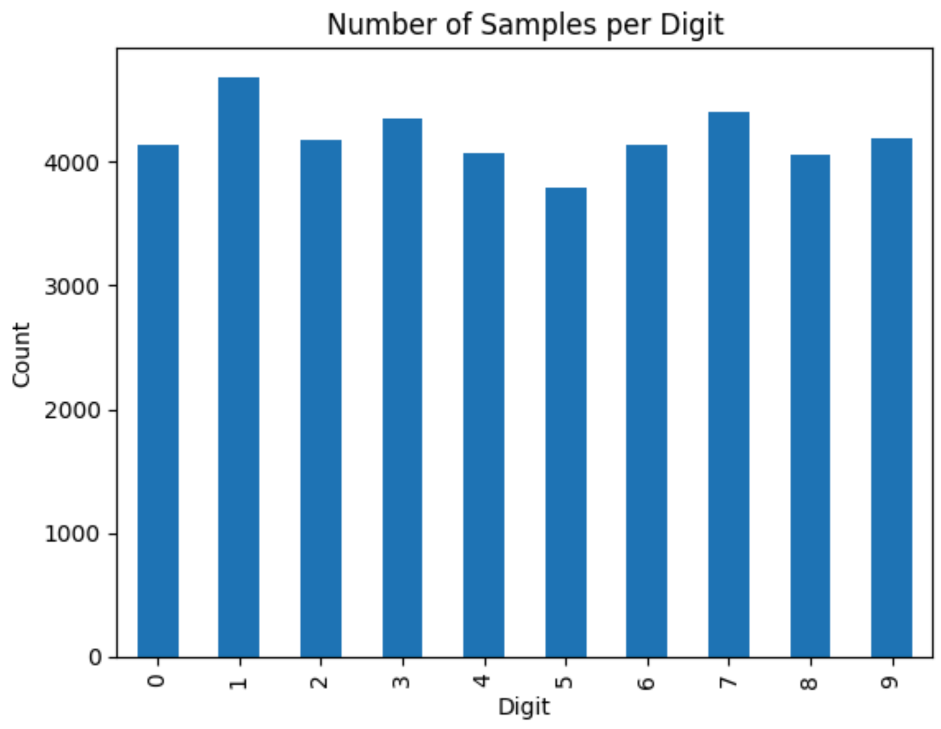


Figure 2

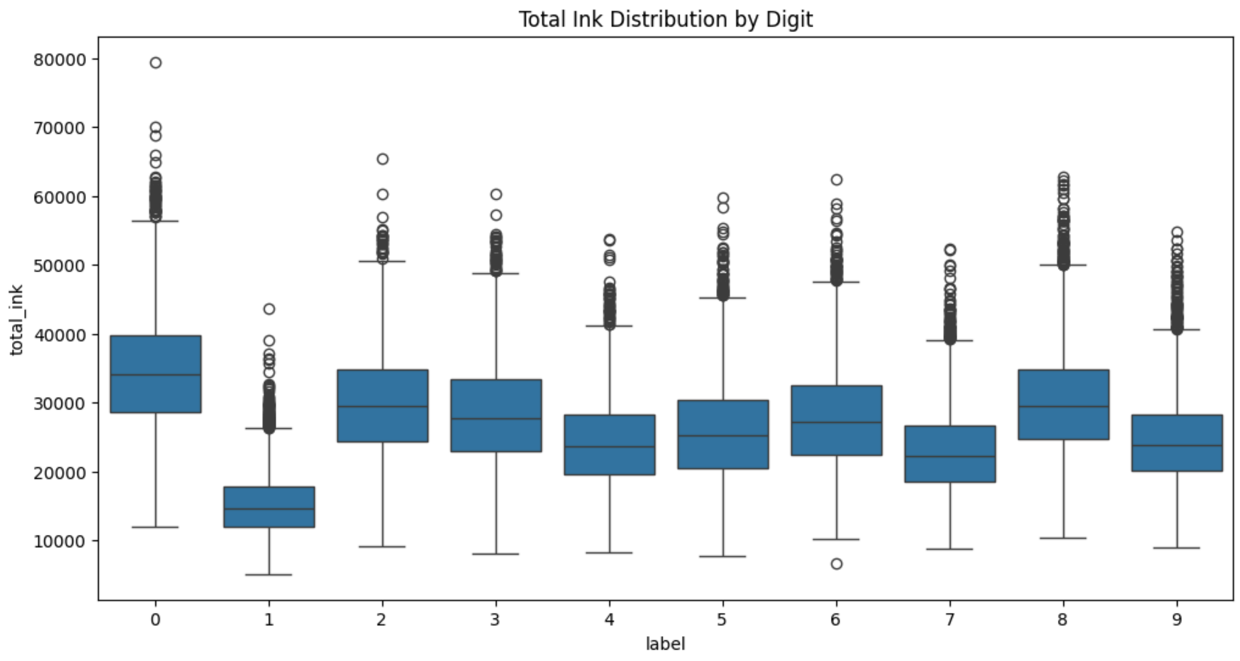
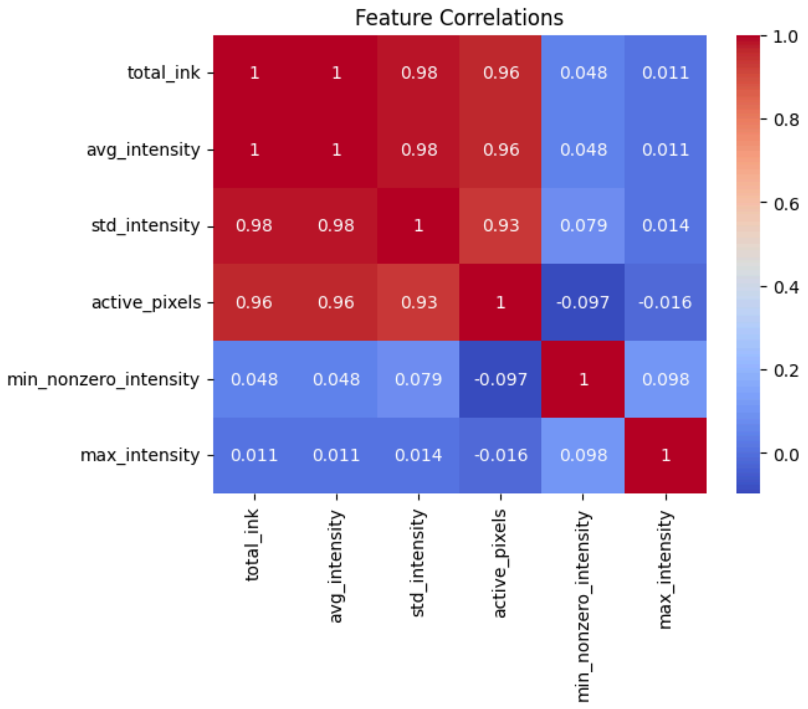


Figure 3



	total_ink	avg_intensity	std_intensity	active_pixels	min_nonzero_intensity	max_intensity
total_ink	1	1	0.98	0.96	0.048	0.011
avg_intensity	1	1	0.98	0.96	0.048	0.011
std_intensity	0.98	0.98	1	0.93	0.079	0.014
active_pixels	0.96	0.96	0.93	1	-0.097	-0.016
min_nonzero_intensity	0.048	0.048	0.079	-0.097	1	0.098
max_intensity	0.011	0.011	0.014	-0.016	0.098	1

Figure 4

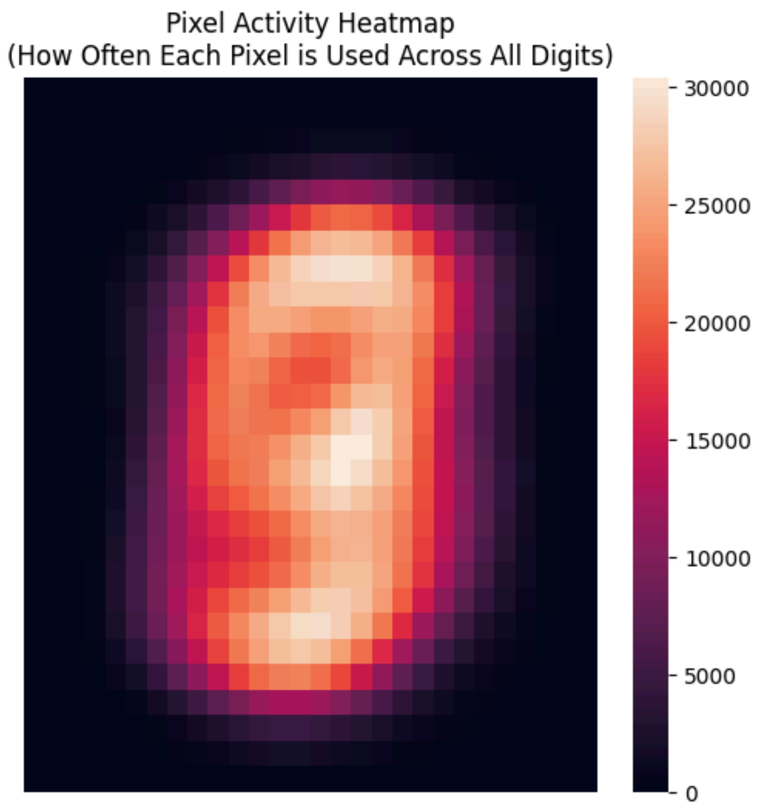


Figure 5

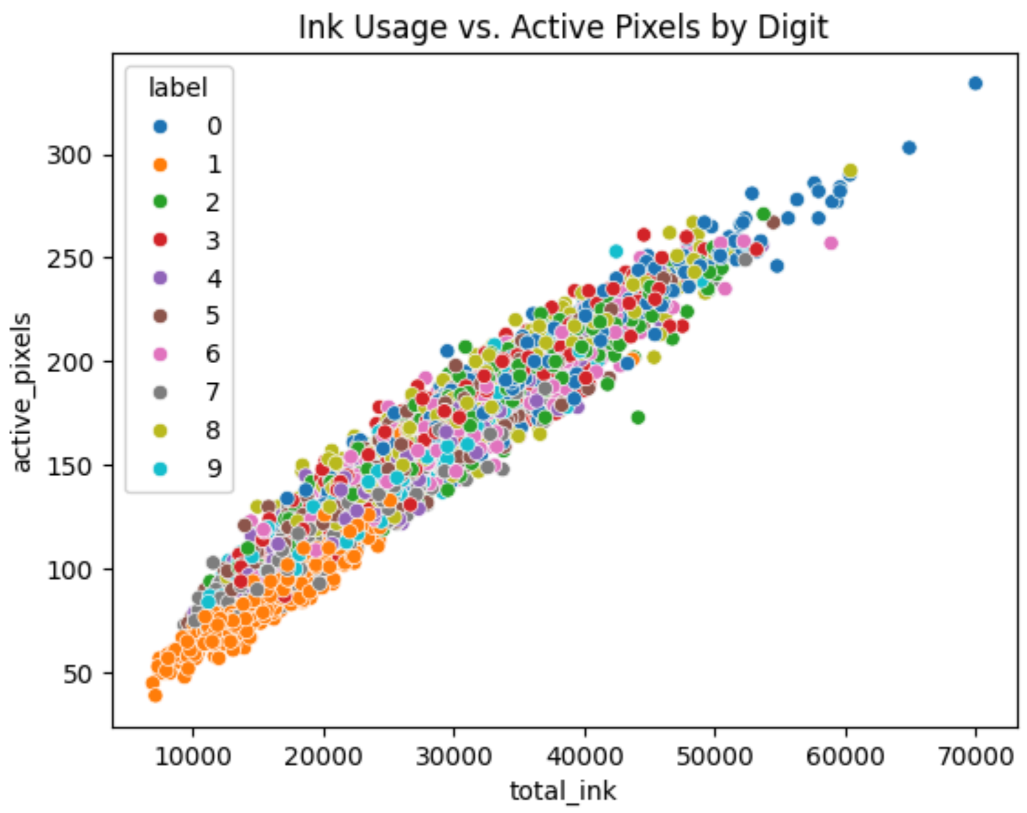


Figure 6

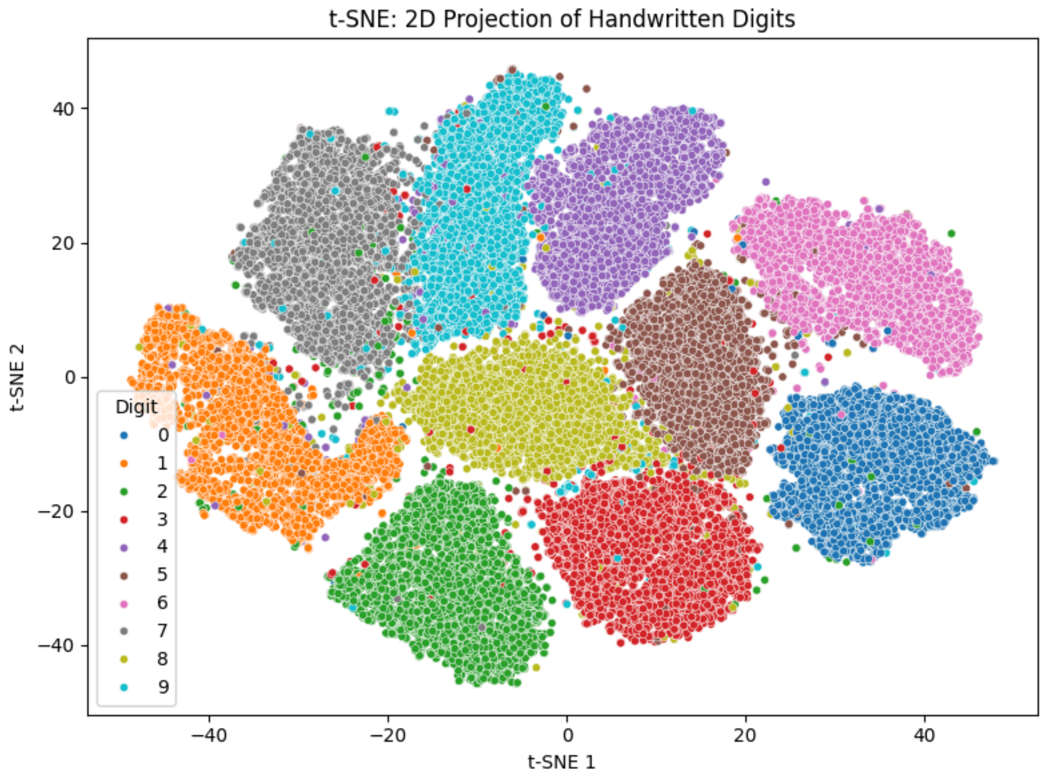


Figure 7

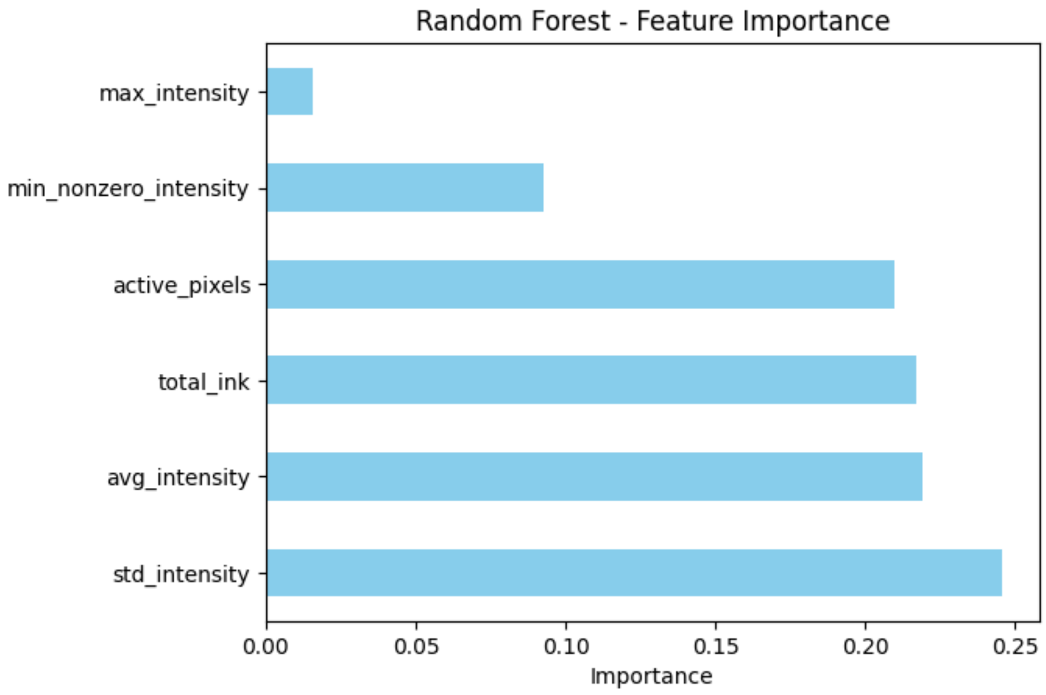


Figure 8

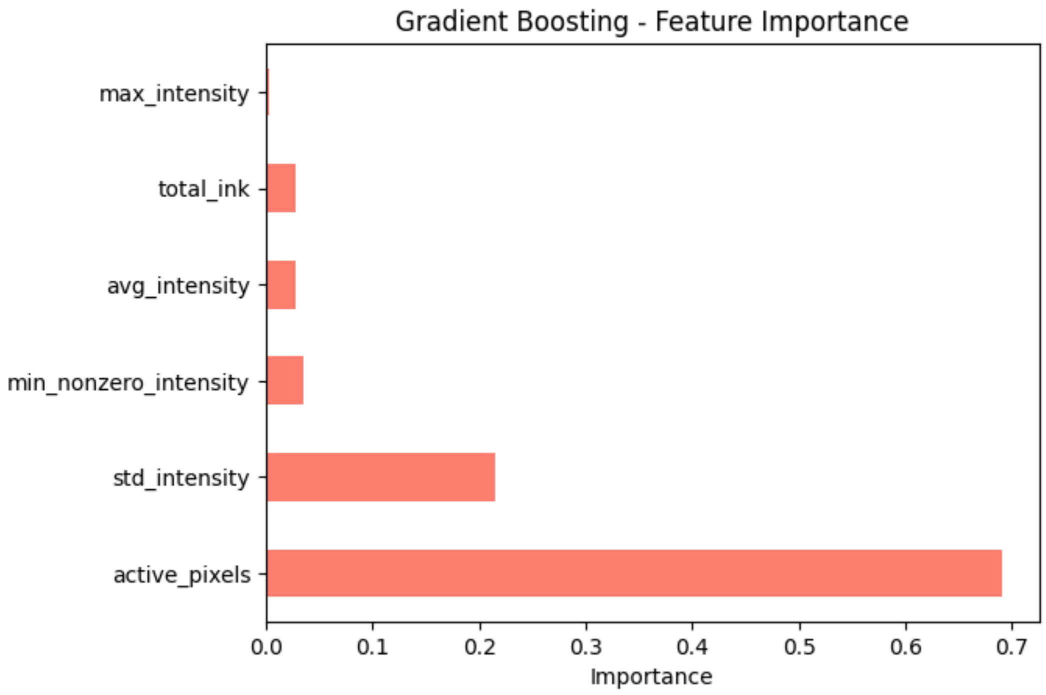
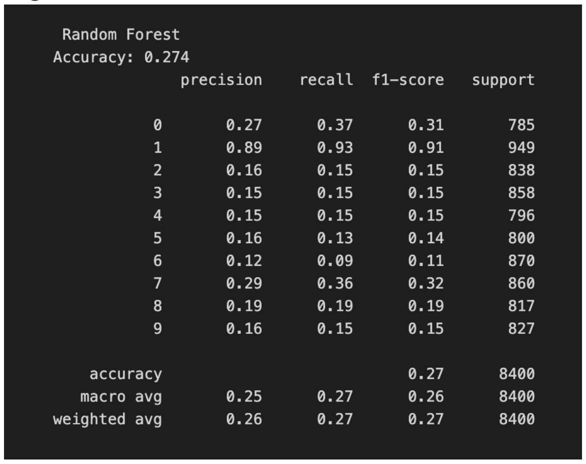
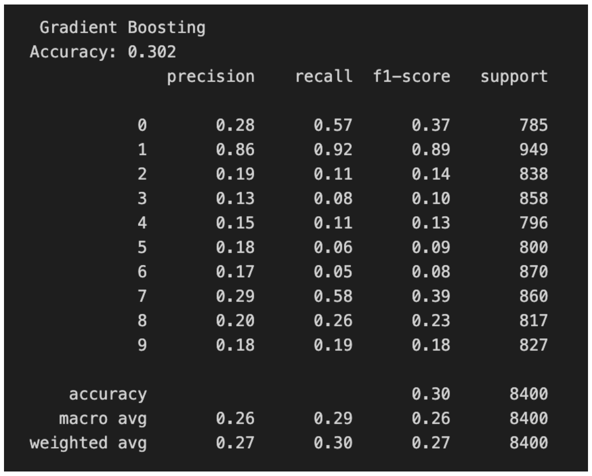


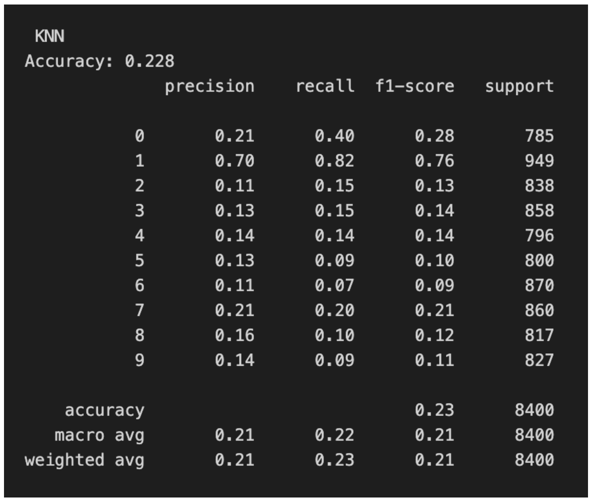
Figure 9:



```
Random Forest
Accuracy: 0.274
              precision    recall  f1-score   support

           0       0.27      0.37      0.31       785
           1       0.89      0.93      0.91       949
           2       0.16      0.15      0.15       838
           3       0.15      0.15      0.15       858
           4       0.15      0.15      0.15       796
           5       0.16      0.13      0.14       800
           6       0.12      0.09      0.11       870
           7       0.29      0.36      0.32       860
           8       0.19      0.19      0.19       817
           9       0.16      0.15      0.15       827

    accuracy                           0.27      8400
   macro avg       0.25      0.27      0.26      8400
weighted avg       0.26      0.27      0.27      8400
```

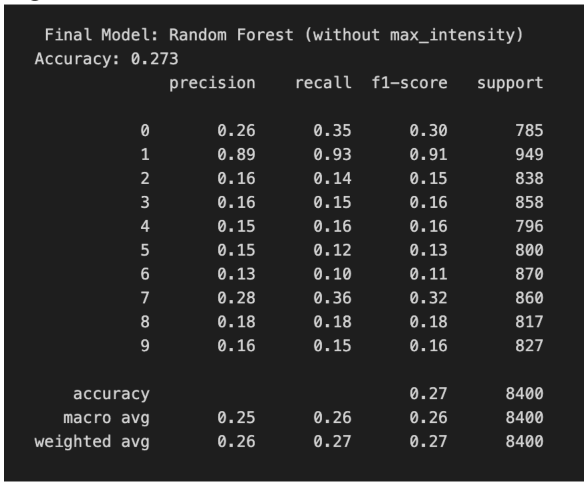


```
Gradient Boosting
Accuracy: 0.302
              precision    recall  f1-score   support

           0       0.28      0.57      0.37       785
           1       0.86      0.92      0.89       949
           2       0.19      0.11      0.14       838
           3       0.13      0.08      0.10       858
           4       0.15      0.11      0.13       796
           5       0.18      0.06      0.09       800
           6       0.17      0.05      0.08       870
           7       0.29      0.58      0.39       860
           8       0.20      0.26      0.23       817
           9       0.18      0.19      0.18       827

    accuracy                           0.30      8400
   macro avg       0.26      0.29      0.26      8400
weighted avg       0.27      0.30      0.27      8400
```

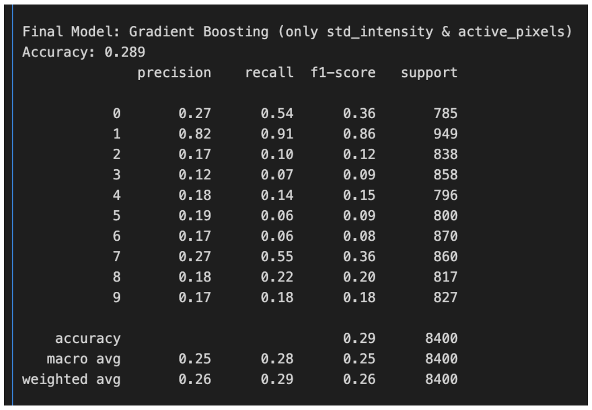


```
KNN
Accuracy: 0.228
              precision    recall  f1-score   support

           0       0.21      0.40      0.28       785
           1       0.70      0.82      0.76       949
           2       0.11      0.15      0.13       838
           3       0.13      0.15      0.14       858
           4       0.14      0.14      0.14       796
           5       0.13      0.09      0.10       800
           6       0.11      0.07      0.09       870
           7       0.21      0.20      0.21       860
           8       0.16      0.10      0.12       817
           9       0.14      0.09      0.11       827

    accuracy                           0.23      8400
   macro avg       0.21      0.22      0.21      8400
weighted avg       0.21      0.23      0.21      8400
```

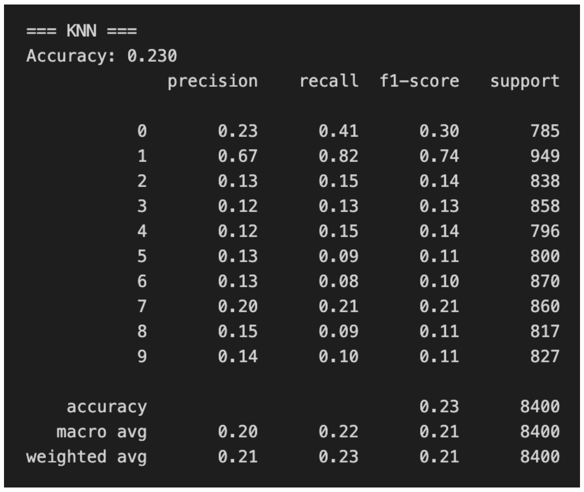
Figure 10:



```
 Final Model: Random Forest (without max_intensity)
Accuracy: 0.273
              precision    recall  f1-score   support

           0       0.26      0.35      0.30       785
           1       0.89      0.93      0.91       949
           2       0.16      0.14      0.15       838
           3       0.16      0.15      0.16       858
           4       0.15      0.16      0.16       796
           5       0.15      0.12      0.13       800
           6       0.13      0.10      0.11       870
           7       0.28      0.36      0.32       860
           8       0.18      0.18      0.18       817
           9       0.16      0.15      0.16       827

    accuracy                           0.27      8400
   macro avg       0.25      0.26      0.26      8400
weighted avg       0.26      0.27      0.27      8400
```

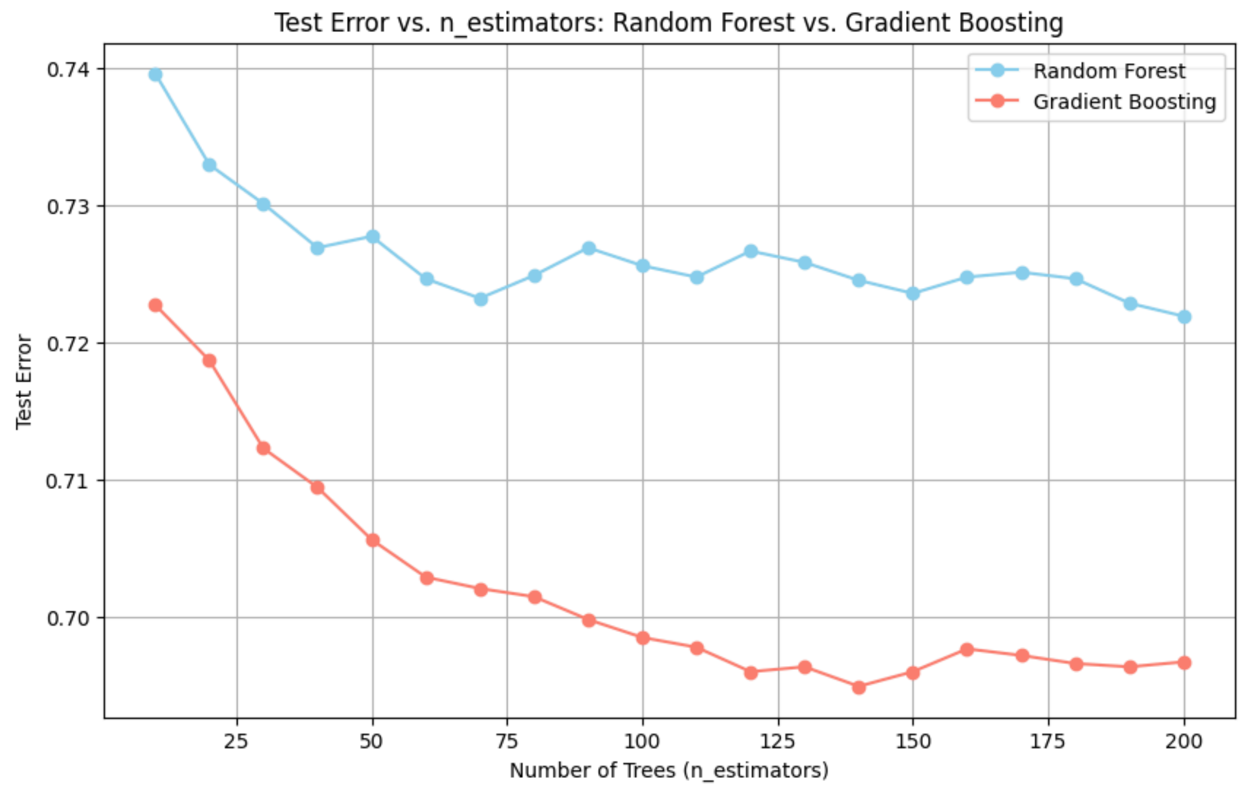


```
Final Model: Gradient Boosting (only std_intensity & active_pixels)
Accuracy: 0.289
              precision    recall  f1-score   support

           0       0.27      0.54      0.36       785
           1       0.82      0.91      0.86       949
           2       0.17      0.10      0.12       838
           3       0.12      0.07      0.09       858
           4       0.18      0.14      0.15       796
           5       0.19      0.06      0.09       800
           6       0.17      0.06      0.08       870
           7       0.27      0.55      0.36       860
           8       0.18      0.22      0.20       817
           9       0.17      0.18      0.18       827

    accuracy                           0.29      8400
   macro avg       0.25      0.28      0.25      8400
weighted avg       0.26      0.29      0.26      8400
```



```
=== KNN ===
Accuracy: 0.230
              precision    recall  f1-score   support

           0       0.23      0.41      0.30       785
           1       0.67      0.82      0.74       949
           2       0.13      0.15      0.14       838
           3       0.12      0.13      0.13       858
           4       0.12      0.15      0.14       796
           5       0.13      0.09      0.11       800
           6       0.13      0.08      0.10       870
           7       0.20      0.21      0.21       860
           8       0.15      0.09      0.11       817
           9       0.14      0.10      0.11       827

    accuracy                           0.23      8400
   macro avg       0.20      0.22      0.21      8400
weighted avg       0.21      0.23      0.21      8400
```

Figure 11:


Test Error vs. n_estimators: Random Forest vs. Gradient Boosting
Random Forest
Gradient Boosting
Test Error
0.74
0.73
0.72
0.71
0.70
25
50
75
100
125
150
175
200
Number of Trees (n_estimators)

In [ ]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns

from sklearn.ensemble import RandomForestClassifier, GradientBoostingClassif
from sklearn.neighbors import KNeighborsClassifier
from sklearn.model_selection import train_test_split, GridSearchCV
from sklearn.metrics import accuracy_score, classification_report, confusion

from sklearn.decomposition import PCA
from sklearn.manifold import TSNE
```

In [ ]:
```
df = pd.read_csv('train.csv')
df.shape
```

420000 observations, 785 columns

In [ ]:
```
df.dtypes
```

In [ ]:
```
df
```

In [ ]:
```
# Identify pixel columns
pixel_columns = [f"pixel{i}" for i in range(784)]

# Create a binary matrix: 1 if pixel > 0, else 0
pixel_activity = (df[pixel_columns] > 0).sum(axis=0).values.reshape(28, 28)

# Plot heatmap of active pixel frequencies
plt.figure(figsize=(6, 6))
sns.heatmap(pixel_activity)
plt.title("Pixel Activity Heatmap\n(How Often Each Pixel is Used Across All
plt.axis("off")
plt.show()
```

Variable Creation

In [ ]:
```
# Identify pixel columns
pixel_columns = [f"pixel{i}" for i in range(784)]

df["total_ink"] = df[pixel_columns].sum(axis=1)
df["avg_intensity"] = df[pixel_columns].mean(axis=1)
df["std_intensity"] = df[pixel_columns].std(axis=1)
df["active_pixels"] = (df[pixel_columns] > 0).sum(axis=1)
df["min_nonzero_intensity"] = df[pixel_columns].replace(0, pd.NA).min(axis=1
df["max_intensity"] = df[pixel_columns].max(axis=1)

df = df[["total_ink", "avg_intensity", "std_intensity",
         "active_pixels", "min_nonzero_intensity", "max_intensity", "label"]

df
```

EDA

In [ ]:
```
df['label'].value_counts().sort_index().plot(kind='bar')
plt.title('Number of Samples per Digit')
plt.xlabel('Digit')
plt.ylabel('Count')
plt.show()
```

In [ ]:
```
plt.figure(figsize=(12, 6))
sns.boxplot(x='label', y='total_ink', data=df)
plt.title('Total Ink Distribution by Digit')
plt.show()
```

In [ ]:
```
sns.heatmap(df[["total_ink", "avg_intensity", "std_intensity", "active_pixel
plt.title('Feature Correlations')
plt.show()
```

In [ ]:
```
sns.scatterplot(x="total_ink", y="active_pixels", hue="label", data=df, pale
plt.title("Ink Usage vs. Active Pixels by Digit")
plt.show()
```

In [ ]:
```
df = pd.read_csv("train.csv")

pixel_columns = [f"pixel{i}" for i in range(784)]
X = df[pixel_columns].values
y = df["label"].values

tsne = TSNE(n_components=2, perplexity=30, n_iter=500, random_state=123)
X_tsne = tsne.fit_transform(X)

plt.figure(figsize=(8, 6))
sns.scatterplot(x=X_tsne[:, 0], y=X_tsne[:, 1], hue=y, palette="tab10", s=20
plt.title("t-SNE: 2D Projection of Handwritten Digits")
plt.xlabel("t-SNE 1")
plt.ylabel("t-SNE 2")
plt.legend(title="Digit")
plt.tight_layout()
plt.show()
```

Base Model

In [ ]:
```
df = pd.read_csv("train.csv")
pixel_columns = [col for col in df.columns if col.startswith("pixel")]

df["total_ink"] = df[pixel_columns].sum(axis=1)
df["avg_intensity"] = df[pixel_columns].mean(axis=1)
df["std_intensity"] = df[pixel_columns].std(axis=1)
df["active_pixels"] = (df[pixel_columns] > 0).sum(axis=1)
df["min_nonzero_intensity"] = df[pixel_columns].replace(0, pd.NA).min(axis=1
df["max_intensity"] = df[pixel_columns].max(axis=1)

df = df[["total_ink", "avg_intensity", "std_intensity",
         "active_pixels", "min_nonzero_intensity", "max_intensity", "label"]
```

```
feature_cols = ["total_ink", "avg_intensity", "std_intensity",
                 "active_pixels", "min_nonzero_intensity", "max_intensity"]
X = df[feature_cols]
y = df["label"]

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, rar

models = {
    "Random Forest": RandomForestClassifier(n_estimators=100, random_state=1
    "Gradient Boosting": GradientBoostingClassifier(n_estimators=100, random
    "KNN": KNeighborsClassifier(n_neighbors=5)
}

# Train and evaluate models
results = {}
for name, model in models.items():
    model.fit(X_train, y_train)
    y_pred = model.predict(X_test)

    acc = accuracy_score(y_test, y_pred)
    results[name] = acc

    print(f"\n {name} ")
    print(f"Accuracy: {acc:.3f}")
    print(classification_report(y_test, y_pred))

    ConfusionMatrixDisplay(confusion_matrix(y_test, y_pred), display_labels=
    plt.title(f"{name} - Confusion Matrix")
    plt.show()

print("Model Comparison Summary")
for name, acc in results.items():
    print(f"{name}: {acc:.3f}")
```

Feature Importance

In [ ]:

```
feature_cols = ["total_ink", "avg_intensity", "std_intensity", "active_pixel
X = df[feature_cols]
y = df["label"]

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, rar

# Feature importance for Random Forest
rf_model = RandomForestClassifier(n_estimators=100, random_state=123)
rf_model.fit(X_train, y_train)

rf_importance = pd.Series(rf_model.feature_importances_, index=feature_cols)
print("Random Forest Feature Importance:")
print(rf_importance)

rf_importance.plot(kind="barh", title="Random Forest - Feature Importance",
plt.xlabel("Importance")
plt.show()
```

```
# Feature Importance for Gradient Boosting
gb_model = GradientBoostingClassifier(n_estimators=100, random_state=123)
gb_model.fit(X_train, y_train)

gb_importance = pd.Series(gb_model.feature_importances_, index=feature_cols)
print("Gradient Boosting Feature Importance:")
print(gb_importance)

gb_importance.plot(kind="barh", title="Gradient Boosting - Feature Importanc
plt.xlabel("Importance")
plt.show()
```

Modeling

Random Forest (w/o Max Intensity) recommended by feature importance

In [ ]:
```
# Features without max_intensity
feature_cols_rf = ["total_ink", "avg_intensity", "std_intensity", "active_pi
X_rf = df[feature_cols_rf]

X_train_rf, X_test_rf, y_train_rf, y_test_rf = train_test_split(X_rf, y, tes

rf_model = RandomForestClassifier(n_estimators=100, random_state=123)
rf_model.fit(X_train_rf, y_train_rf)

y_pred_rf = rf_model.predict(X_test_rf)

acc_rf = accuracy_score(y_test_rf, y_pred_rf)
print("\n Random Forest (without max_intensity)")
print(f"Accuracy: {acc_rf:.3f}")
print(classification_report(y_test_rf, y_pred_rf))
```

Gradient Boosting Recommended by Feature Importance

In [ ]:
```
feature_cols_gb = ["std_intensity", "active_pixels"]
X_gb = df[feature_cols_gb]

X_train_gb, X_test_gb, y_train_gb, y_test_gb = train_test_split(X_gb, y, tes

gb_model = GradientBoostingClassifier(n_estimators=100, random_state=123)
gb_model.fit(X_train_gb, y_train_gb)

y_pred_gb = gb_model.predict(X_test_gb)

acc_gb = accuracy_score(y_test_gb, y_pred_gb)
print("\nGradient Boosting (only std_intensity & active_pixels)")
print(f"Accuracy: {acc_gb:.3f}")
print(classification_report(y_test_gb, y_pred_gb))
```

Hyperparamter Tuning

Finding the best n_estimator for RandomForest and Gradient Boosting

```
In [ ]: errors_rf = []
        n_values = range(10, 201, 10)

        for n in n_values:
            rf_model = RandomForestClassifier(n_estimators=n, random_state=123)
            rf_model.fit(X_train, y_train)
            errors_rf.append(1 - rf_model.score(X_test, y_test))

        errors_gb = []
        for n in n_values:
            gb_model = GradientBoostingClassifier(n_estimators=n, random_state=123)
            gb_model.fit(X_train, y_train)
            errors_gb.append(1 - gb_model.score(X_test, y_test))

        plt.figure(figsize=(10, 6))
        plt.plot(n_values, errors_rf, marker='o', color='skyblue', label="Random For
        plt.plot(n_values, errors_gb, marker='o', color='salmon', label="Gradient Bo
        plt.xlabel("Number of Trees (n_estimators)")
        plt.ylabel("Test Error")
        plt.title("Test Error vs. n_estimators: Random Forest vs. Gradient Boosting"
        plt.legend()
        plt.grid(True)
        plt.show()

        min_error_rf = min(errors_rf)
        best_n_rf = n_values[errors_rf.index(min_error_rf)]
        print(f"Random Forest: Lowest Test Error = {min_error_rf:.4f} at n_estimator

        min_error_gb = min(errors_gb)
        best_n_gb = n_values[errors_gb.index(min_error_gb)]
        print(f"Gradient Boosting: Lowest Test Error = {min_error_gb:.4f} at n_estim
```

KNN finding the best K

```
In [ ]: # Grid search for best k
        param_grid_knn = {'n_neighbors': list(range(1, 21))}  # Try k from 1 to 20

        grid_search_knn = GridSearchCV(KNeighborsClassifier(),
                                       param_grid_knn,
                                       cv=5,
                                       scoring='accuracy',
                                       n_jobs=-1)

        grid_search_knn.fit(X_train, y_train)

        # Best k
        best_k = grid_search_knn.best_params_['n_neighbors']
        best_acc = grid_search_knn.best_score_

        print(f"Best k (n_neighbors) for KNN: {best_k}")
        print(f"Cross-validated accuracy: {best_acc:.4f}")
```

Build model using the selected n_estimators and K

Final Model: Random Forest

In [ ]:
```
# Features without max_intensity
feature_cols_rf = ["total_ink", "avg_intensity", "std_intensity", "active_pi
X_rf = df[feature_cols_rf]

X_train_rf, X_test_rf, y_train_rf, y_test_rf = train_test_split(X_rf, y, tes

rf_model = RandomForestClassifier(n_estimators=best_n_rf, random_state=123)
rf_model.fit(X_train_rf, y_train_rf)

y_pred_rf = rf_model.predict(X_test_rf)

acc_rf = accuracy_score(y_test_rf, y_pred_rf)
print("\n Final Model: Random Forest (without max_intensity)")
print(f"Accuracy: {acc_rf:.3f}")
print(classification_report(y_test_rf, y_pred_rf))
```

Final Model: Gradient Boosting

In [ ]:
```
feature_cols_gb = ["std_intensity", "active_pixels"]
X_gb = df[feature_cols_gb]

X_train_gb, X_test_gb, y_train_gb, y_test_gb = train_test_split(X_gb, y, tes

gb_model = GradientBoostingClassifier(n_estimators=best_n_gb, random_state=1
gb_model.fit(X_train_gb, y_train_gb)

y_pred_gb = gb_model.predict(X_test_gb)

acc_gb = accuracy_score(y_test_gb, y_pred_gb)
print("\nFinal Model: Gradient Boosting (only std_intensity & active_pixels)
print(f"Accuracy: {acc_gb:.3f}")
print(classification_report(y_test_gb, y_pred_gb))
```

In [ ]:
```
feature_cols = ["total_ink", "avg_intensity", "std_intensity", "active_pixel
X = df[feature_cols]
y = df["label"]

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, rar

models = {
    "KNN": KNeighborsClassifier(n_neighbors=best_k)
}

results = {}
for name, model in models.items():
    model.fit(X_train, y_train)
    y_pred = model.predict(X_test)

    acc = accuracy_score(y_test, y_pred)
    results[name] = acc

    print(f"\n=== {name} ===")
    print(f"Accuracy: {acc:.3f}")
    print(classification_report(y_test, y_pred))

    ConfusionMatrixDisplay(confusion_matrix(y_test, y_pred), display_labels=
    plt.title(f"{name} - Confusion Matrix")
    plt.show()

print("\nModel Comparison Summary")
for name, acc in results.items():
    print(f"{name}: {acc:.3f}")
```