

# BIM309- Artificial Intelligence- Term Project

## Odometer Classification and Mileage Extraction using Machine Learning

YUNUS EMRE KORKMAZ- 57847323212

### *Abstract:*

**This study delves into the domain of machine learning applications in the context of odometer analysis, focusing on the TRODO dataset [1] containing 2389 annotated odometer images. Task 1 involves the classification of odometer types, distinguishing between analog and digital variants, while Task 2 aims at extracting mileage information from these images. Dataset exploration reveals a diverse collection of annotated odometers, easing an in-depth examination of both classification and extraction tasks.**

**Challenges met in both tasks are discussed, paving the way for potential improvements. The findings supply valuable insights into the feasibility and effectiveness of machine learning in odometer analysis. This study contributes to the growing body of research in computer vision applications in the automotive domain.**

*KEYWORDS: ODOMETER CLASSIFICATION, MILEAGE EXTRACTION, MACHINE LEARNING, COMPUTER VISION, DATASET ANALYSIS.*

# INTRODUCTION:

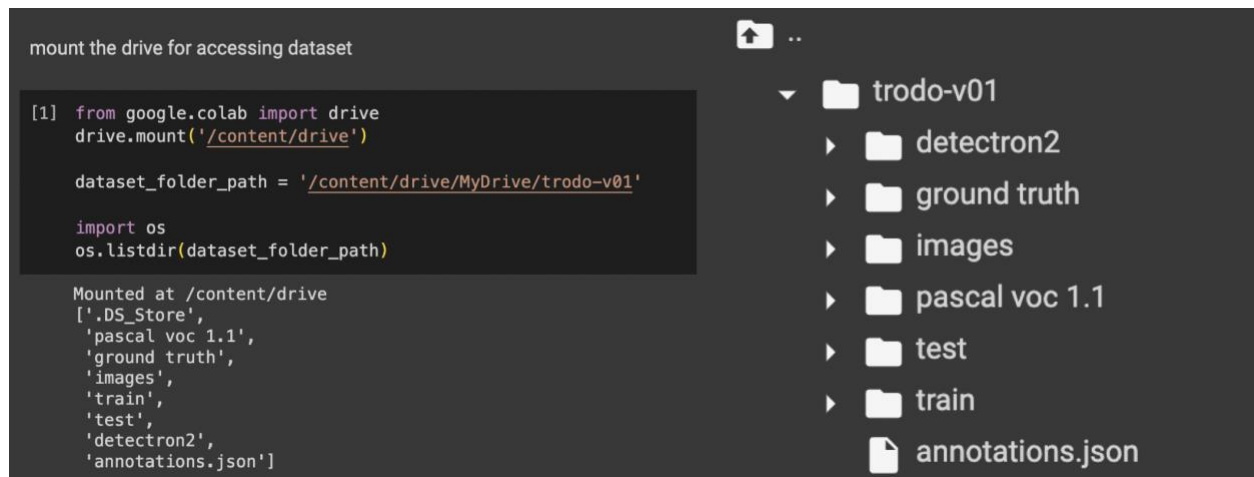
While doing this task, I used Google Colab notebook and saved the dataset to my Drive and accessed the dataset from there. The reason for this was to prevent local files from being lost and reloading the same files in case of a Runtime crash. I am attaching this notebook's link [here](#) but, it is not possible to run this code without connecting to my Drive.

## Task 1: Odometer Type Classification (Analog or Digital)

The general steps:

### 1.1 Getting the dataset:

I download the dataset and upload it to my Drive to keep it safe and prevent from reupload same folder repeatedly. The folder structure of notebook is shown in *Figure 1*.



*Figure 1.1: The folder structure of notebook and dataset*

## 1.2 Dataset Exploration

The images consist of both digital and mechanical/analog odometers. In the dataset there are 858 analog odometers and 1531 digital images. When we refer to digital odometers, it means that the digits are presented electronically on a digital screen which can be seen in figure 3.1 b. Analog odometer means that the odometer present the digits on a mechanical roll which can be seen in figure 3.1 a. The images have a high variance in the parameter's illumination, resolution and rotation and the dashboards are from different models and car manufacturers. The labeling of the data contains odometer types, position of the odometers and the mileage. The mileage meaning how many kilometers the car has driven.[2]

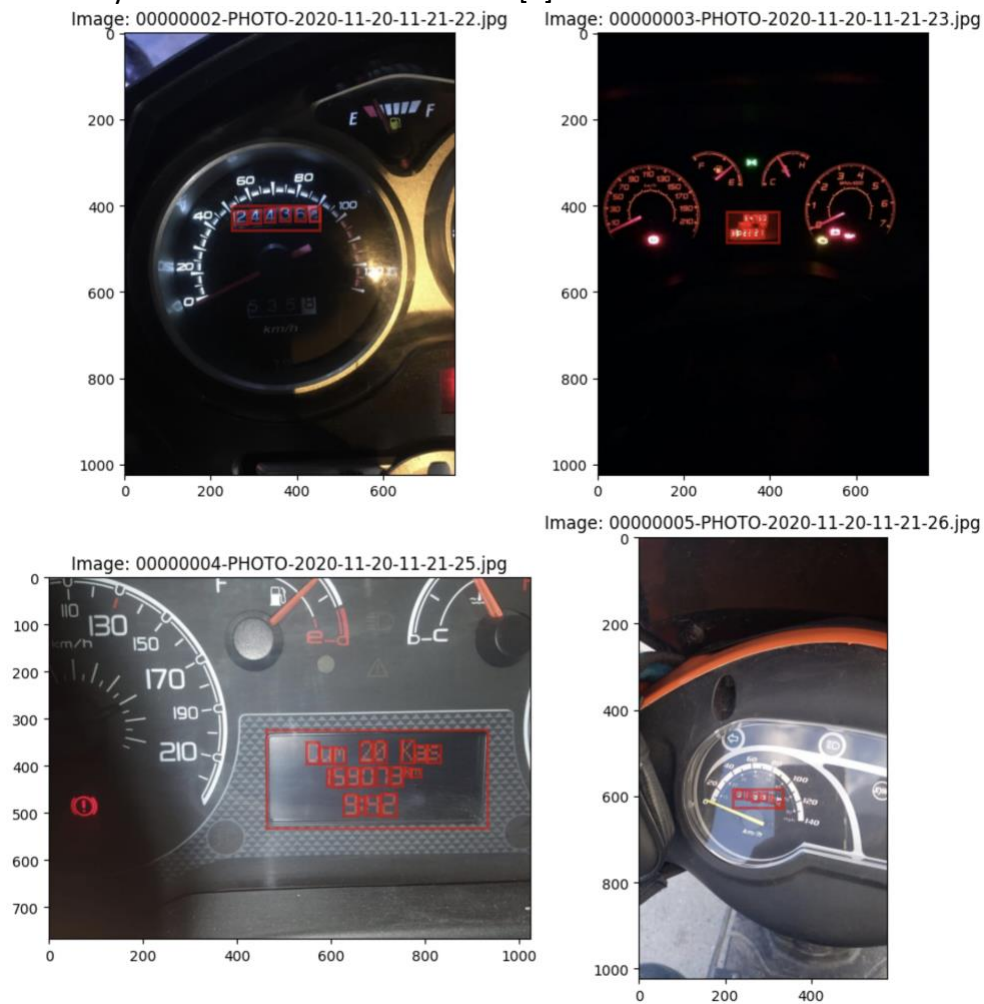
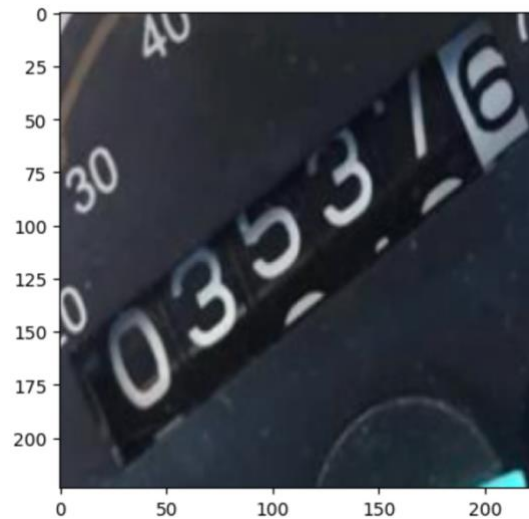


Figure 1.2: Sample images from dataset with bounding boxes of odometer and digits

### 1.3 Data Preprocessing

The only and important part of those images on this task is odometer bounding boxes. So, I firstly take the bounding box coordinates of images from corresponding xml file and then crop image. Then I resize the image to 224x224 and then normalize it. Detailed implementation and source code are available in the notebook.



*Figure 1.3a Resized cropped and normalized image from dataset.*

I also convert the labels of images from string labels to numeric labels to prevent calculation errors that could occur in future steps.

```
[ ] label_mapping = dict(zip(label_encoder.classes_, label_encoder.transform(label_encoder.classes_)))  
print("Label Mapping:", label_mapping)  
  
Label Mapping: {'analog': 0, 'digital': 1}
```

*Figure 1.3b: Resized cropped and normalized image from dataset*

I also convert lists to NumPy arrays for better compatibility with Keras and other ML models.

### ▼ Label Encodings

```
[ ] from sklearn.preprocessing import LabelEncoder
import numpy as np

label_encoder = LabelEncoder()
y_train_numeric = label_encoder.fit_transform(y_train)

X_train_array = np.array(X_train)
y_train_array = np.array(y_train_numeric)

y_test_numeric = label_encoder.transform(y_test)
X_test_array = np.array(X_test)
y_test_array = np.array(y_test_numeric)
```

*Figure 1.3c: Label Encodings and NumPy array transformations.*

Finally, to avoid re-running the preprocessing cell every time I write a python script that saves the output.

```
[ ] import pickle
from google.colab import drive

drive.mount('/content/drive')

# Save the processed data to Google Drive
file_path = '/content/drive/MyDrive/processed_data.pkl'
#with open(file_path, 'wb') as f:
#    pickle.dump(X, f)
#    pickle.dump(y, f)

# To load the processed data in a future session
with open(file_path, 'rb') as f:
    X = pickle.load(f)
    y = pickle.load(f)
```

*Figure 1.3d: Saving and loading the preprocessed data for future sessions.*

## 1.4 Machine Learning Algorithms

### 1.4.1) KNeighbors Classifier

The KNeighbors Classifier is a simple yet effective algorithm that classifies data points based on the majority class of their k-nearest neighbors. In our context, it can capture local patterns in the dataset, making it suitable for discerning subtle differences between analog and digital odometers.

I chose KNeighbors Classifier due to its ease of implementation and ability to capture intricate relationships within the data, which could be valuable for distinguishing between the visual features of analog and digital odometers.

### 1.4.2) Random Forest Classifier

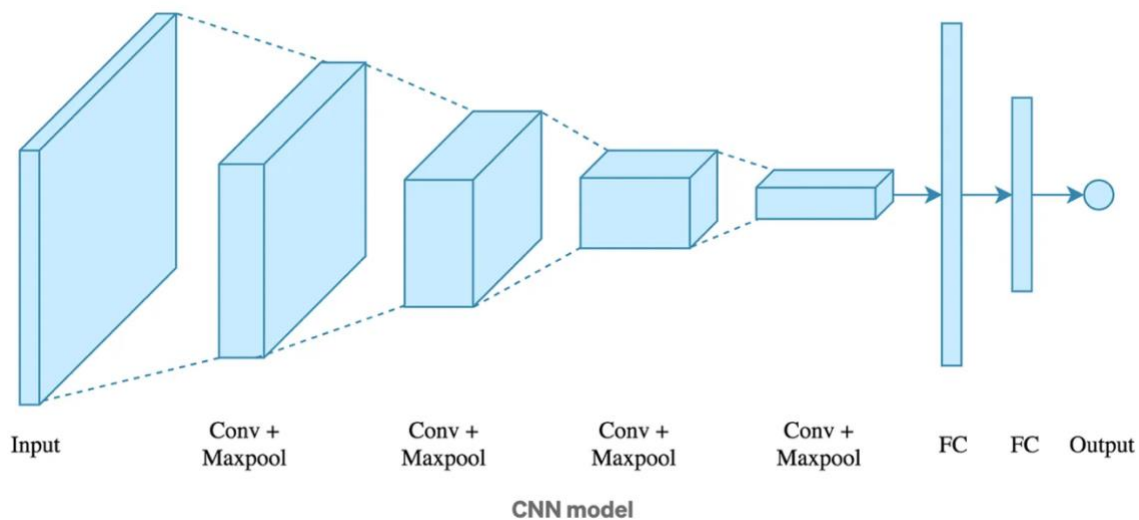
A Random Forest Classifier operates by constructing a multitude of decision trees and outputs the mode of the classes (classification) or mean prediction (regression) of individual trees. This ensemble method is robust and less prone to overfitting.

I chose this because of its ability to handle complex relationships in the data, supplying a robust solution for odometer type classification. Its ensemble nature enables it to mitigate noise and enhance accuracy.

### 1.4.3) Convolutional Neural Network (CNN)

CNNs (Convolutional Neural Networks) are a category of deep neural networks specifically designed for processing structured grid data, such as images. They use convolutional layers to automatically learn hierarchical representations of features.

The visual nature of odometer images makes CNNs well-suited for feature extraction. By using the hierarchical representations learned through convolutional layers, a CNN can effectively discern intricate patterns in the images, enhancing the accuracy of odometer type classification.



*Figure 1..3: A simple CNN model used for binary classification.*

(Image is captured from: <https://medium.com/@mayankverma05032001/binary-classification-using-convolution-neural-network-cnn-model-6e35cdf5bdbb>)

## 1.5 Model Training and Evaluation

I split the train and test set by %80 for training and %20 for testing. Also, I set random state to 42 to get reproducible outputs and prevent random outputs.

```
[ ] from sklearn.model_selection import train_test_split

    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
```

Figure 1.5a: Splitting training and test sets.

I also flatten each image on the train set for compatibility with algorithms and simplification of input.

Flatten each image on train set

```
[ ] X_train_flat = np.reshape(X_train_array, (X_train_array.shape[0], -1))
    X_test_flat = np.reshape(X_test_array, (X_test_array.shape[0], -1))
```

Figure 1.5b: Flatten each image on dataset.

Detailed implementations of each algorithm can be found in notebook.

### 1.5.1) KNeighbors Classifier

▼ KNeighborsClassifier

▶

```
from sklearn.neighbors import KNeighborsClassifier
from tqdm.notebook import tqdm
import numpy as np

# Initialize the KNeighborsClassifier
knn_clf = KNeighborsClassifier()

# Fit the model with tqdm progress bar
for epoch in tqdm(range(10), desc="Fitting KNN", position=0, leave=True):
    knn_clf.fit(X_train_flat, y_train_array)

# Calculate baseline accuracy
baseline_accuracy = knn_clf.score(X_test_flat, y_test_array)
print(f"Baseline Accuracy: {baseline_accuracy:.4f}")
```

🔍

Fitting KNN: 100%  10/10 [00:02<00:00, 4.84it/s]

Baseline Accuracy: 0.8661

Figure 1.5.1a: Implementation of unoptimized KNeighbors classifier

We got 0.8661 accuracy on default KNN. This is not a good accuracy at all. So, I tried to find the best hyperparameters for KNN using GridSearch.

Find the best parameters using GridSearch

```
[ ] from sklearn.model_selection import GridSearchCV

    param_grid = [{'weights': ["uniform", "distance"], 'n_neighbors': [3, 4, 5, 6]}]

    knn_clf = KNeighborsClassifier()
    grid_search = GridSearchCV(knn_clf, param_grid, cv=5)
    grid_search.fit(X_train_flat, y_train_array)
```

```
GridSearchCV
└─ estimator: KNeighborsClassifier
   └─ KNeighborsClassifier
```

```
[ ] grid_search.best_params_

{'n_neighbors': 4, 'weights': 'uniform'}
```

Figure 1.5.1b: Finding the best parameters using GridSearch.

Results with best parameters are shown in Figure 1.4.1c

```
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score

# Assuming grid_search.best_estimator_ is your model
y_pred = grid_search.best_estimator_.predict(X_test_flat)

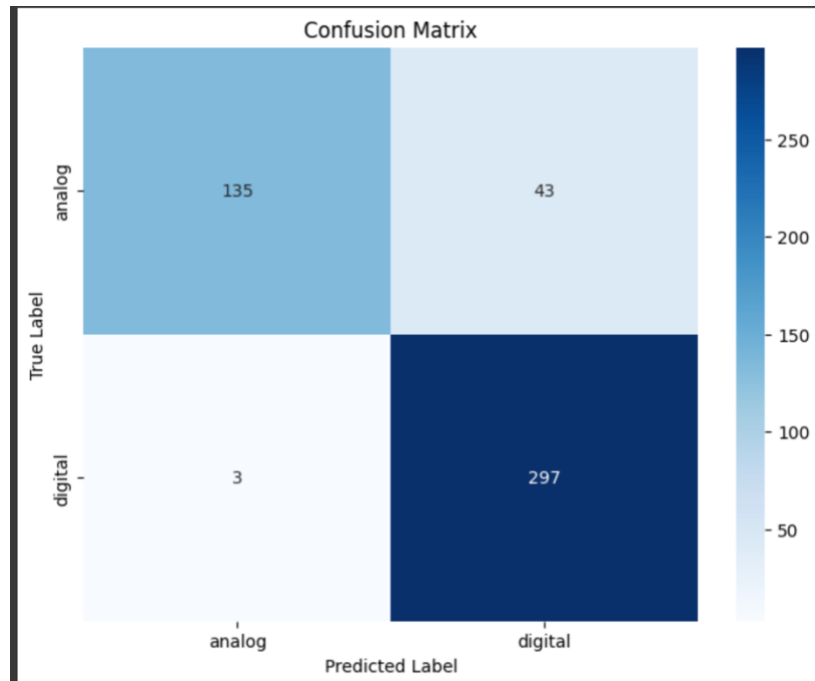
accuracy = accuracy_score(y_test_numeric, y_pred)
precision = precision_score(y_test_numeric, y_pred, average='binary')
recall = recall_score(y_test_numeric, y_pred, average='binary')
f1 = f1_score(y_test_numeric, y_pred, average='binary')

print("Accuracy:", accuracy)
print("Precision:", precision)
print("Recall:", recall)
print("F1 Score:", f1)
```

```
Accuracy: 0.9037656903765691
Precision: 0.8735294117647059
Recall: 0.99
F1 Score: 0.928125
```

Figure 1.5.1c: Accuracy, Precision, Recall and F1 Score of tuned KNN.

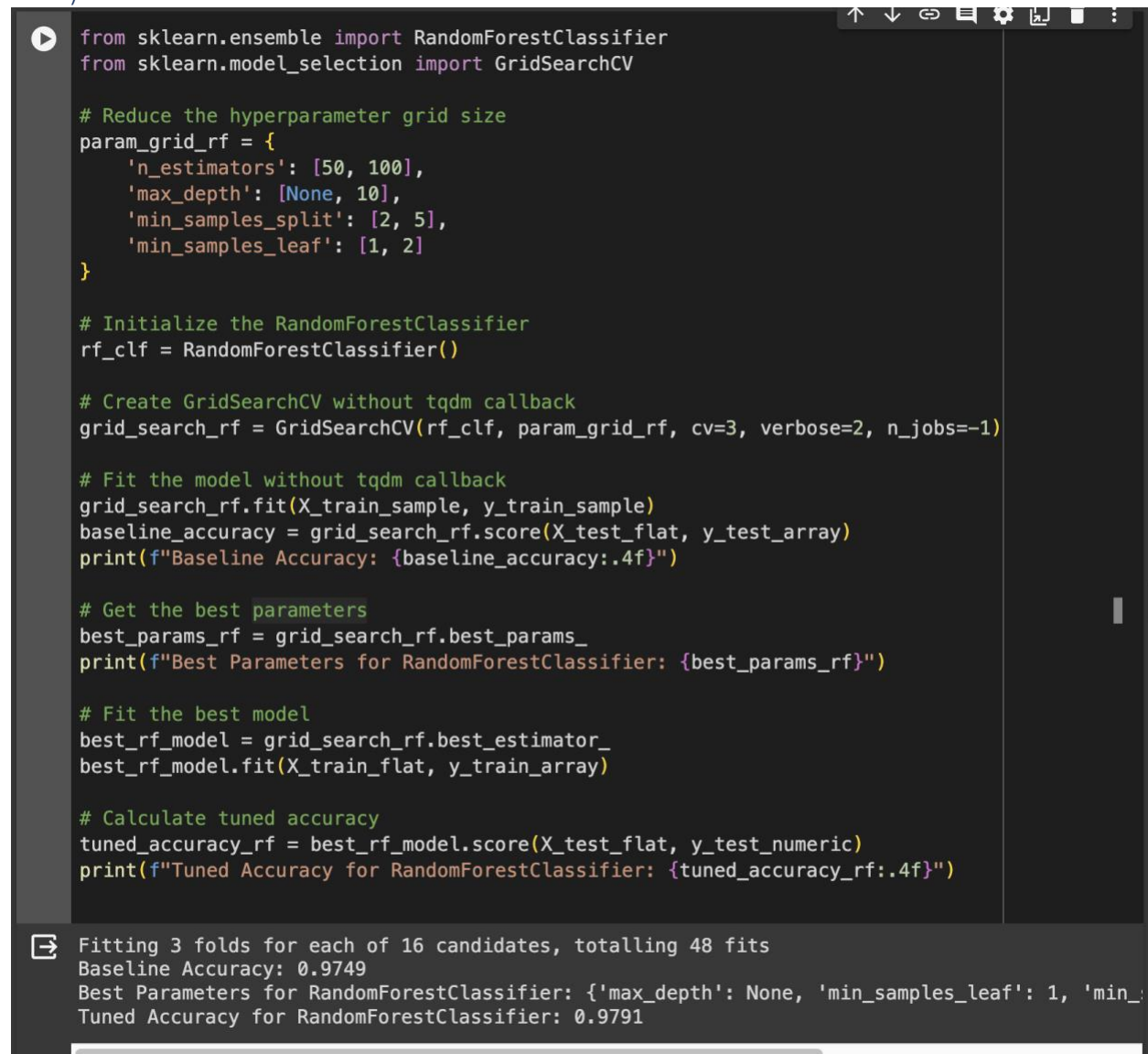




*Figure 1.4.1d: Confusion Matrix with best parameters of KNN*

Our accuracy increased from 0.86 to 0.90 after hyperparameter tuning.

### 1.5.2) Random Forest Classifier



```
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import GridSearchCV

# Reduce the hyperparameter grid size
param_grid_rf = {
    'n_estimators': [50, 100],
    'max_depth': [None, 10],
    'min_samples_split': [2, 5],
    'min_samples_leaf': [1, 2]
}

# Initialize the RandomForestClassifier
rf_clf = RandomForestClassifier()

# Create GridSearchCV without tqdm callback
grid_search_rf = GridSearchCV(rf_clf, param_grid_rf, cv=3, verbose=2, n_jobs=-1)

# Fit the model without tqdm callback
grid_search_rf.fit(X_train_sample, y_train_sample)
baseline_accuracy = grid_search_rf.score(X_test_flat, y_test_array)
print(f"Baseline Accuracy: {baseline_accuracy:.4f}")

# Get the best parameters
best_params_rf = grid_search_rf.best_params_
print(f"Best Parameters for RandomForestClassifier: {best_params_rf}")

# Fit the best model
best_rf_model = grid_search_rf.best_estimator_
best_rf_model.fit(X_train_flat, y_train_array)

# Calculate tuned accuracy
tuned_accuracy_rf = best_rf_model.score(X_test_flat, y_test_numeric)
print(f"Tuned Accuracy for RandomForestClassifier: {tuned_accuracy_rf:.4f}")
```

➡ Fitting 3 folds for each of 16 candidates, totalling 48 fits  
Baseline Accuracy: 0.9749  
Best Parameters for RandomForestClassifier: {'max\_depth': None, 'min\_samples\_leaf': 1, 'min\_...  
Tuned Accuracy for RandomForestClassifier: 0.9791

Figure 1.5.2a: Random Forest Classifier implementation.

```
Accuracy: 0.9791
Precision: 0.9966
Recall: 0.9700
F1 Score: 0.9831
Confusion Matrix:
[[177   1]
 [  9 291]]
```

Figure 1.5.2b Random Forest Classifier Accuracy Precision Recall F1 Score and Confusion Matrix.

Random forest classifier gets a better result than KNN Classifier. I also did hyperparameter tuning but, there is no significant amount of increase in accuracy of model. Nevertheless, %97 accuracy is pretty well and Random forest classifiers is a way better closure to our problem.

### 1.5.3) Convolutional Neural Network (CNN)

```
from tensorflow.keras.models import load_model
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score, confusion_matrix

# Assuming you've defined model as mentioned in the previous code

# Ensure the output layer is configured for binary classification
model = Sequential()
model.add(Conv2D(32, (3, 3), activation='relu', input_shape=(224, 224, 3)))
model.add(MaxPooling2D((2, 2)))
model.add(Conv2D(64, (3, 3), activation='relu'))
model.add(MaxPooling2D((2, 2)))
model.add(Flatten())
model.add(Dense(64, activation='relu'))
model.add(Dense(1, activation='sigmoid')) # Change the number of units to 1 for binary classification

# Compile the model
model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])

# Define callbacks
checkpoint = ModelCheckpoint('best_model.h5', save_best_only=True)
early_stopping = EarlyStopping(monitor='val_loss', patience=3)

# Train the model with callbacks
model.fit(X_train_array, y_train_array, epochs=10, validation_data=(X_test_array, y_test_array), callbacks=[checkpoint, early_stopping])

# Load the best weights after training
best_model = load_model('best_model.h5')

# Make predictions on the test set
y_pred_proba = best_model.predict(X_test_array)
y_pred = (y_pred_proba > 0.5).astype(int).flatten() # Ensure y_pred is a 1D array

# Convert y_test_array to match the binary format
y_test_binary = y_test_array.flatten() # Ensure y_test_binary is a 1D array

# Calculate metrics
accuracy = accuracy_score(y_test_binary, y_pred)
precision = precision_score(y_test_binary, y_pred)
recall = recall_score(y_test_binary, y_pred)
f1 = f1_score(y_test_binary, y_pred)
conf_matrix = confusion_matrix(y_test_binary, y_pred)

# Print the metrics
print("Accuracy:", accuracy)
print("Precision:", precision)
print("Recall:", recall)
print("F1 Score:", f1)
print("Confusion Matrix:\n", conf_matrix)
```

Figure 1.5.3a: CNN implementation

```

Epoch 1/10
58/60 [=====] - ETA: 0s - loss: 0.2662 - accuracy: 0.9025/usr/local/lib/python3.10/dist-packages/keras/src/engine/trai
saving_api.save_model(
60/60 [=====] - 5s 45ms/step - loss: 0.2625 - accuracy: 0.9037 - val_loss: 0.0663 - val_accuracy: 0.9728
Epoch 2/10
60/60 [=====] - 2s 29ms/step - loss: 0.0486 - accuracy: 0.9853 - val_loss: 0.0162 - val_accuracy: 0.9874
Epoch 3/10
60/60 [=====] - 2s 35ms/step - loss: 0.0201 - accuracy: 0.9937 - val_loss: 0.0076 - val_accuracy: 0.9979
Epoch 4/10
60/60 [=====] - 2s 28ms/step - loss: 0.0071 - accuracy: 0.9984 - val_loss: 0.0034 - val_accuracy: 1.0000
Epoch 5/10
60/60 [=====] - 1s 21ms/step - loss: 0.0048 - accuracy: 0.9990 - val_loss: 0.0038 - val_accuracy: 0.9979
Epoch 6/10
60/60 [=====] - 2s 30ms/step - loss: 0.0050 - accuracy: 0.9990 - val_loss: 0.0022 - val_accuracy: 1.0000
Epoch 7/10
60/60 [=====] - 1s 21ms/step - loss: 0.0038 - accuracy: 0.9990 - val_loss: 0.2147 - val_accuracy: 0.9519
Epoch 8/10
60/60 [=====] - 1s 21ms/step - loss: 0.0236 - accuracy: 0.9927 - val_loss: 0.0246 - val_accuracy: 0.9874
Epoch 9/10
60/60 [=====] - 1s 21ms/step - loss: 0.0427 - accuracy: 0.9874 - val_loss: 0.0139 - val_accuracy: 0.9979
15/15 [=====] - 0s 9ms/step
Accuracy: 1.0

```

*Figure 1.5.3b: outputs from best model of CNN.*

As we seen in the outputs we got nearly %100 accuracy on training.

```

15/15 [=====]
Accuracy: 1.0
Precision: 1.0
Recall: 1.0
F1 Score: 1.0
Confusion Matrix:
[[178  0]
 [  0 300]]

```

*Figure 1.5.3c: Accuracy Precision Recall F1 Score and confusion matrix from best model of CNN.*



```
model.load_weights('best_model.h5')  
model.summary()
```



Model: "sequential\_1"

Layer (type)	Output Shape	Param #
conv2d_2 (Conv2D)	(None, 222, 222, 32)	896
max_pooling2d_2 (MaxPooling2D)	(None, 111, 111, 32)	0
conv2d_3 (Conv2D)	(None, 109, 109, 64)	18496
max_pooling2d_3 (MaxPooling2D)	(None, 54, 54, 64)	0
flatten_1 (Flatten)	(None, 186624)	0
dense_2 (Dense)	(None, 64)	11944000
dense_3 (Dense)	(None, 1)	65
Total params: 11963457 (45.64 MB)		
Trainable params: 11963457 (45.64 MB)		
Non-trainable params: 0 (0.00 Byte)		

Figure 1.5.3d: Summary from best model of CNN

## 1.6 Model Comparison

The performance of three distinct models—KNeighbors Classifier, Random Forest Classifier, and Convolutional Neural Network (CNN)—was evaluated for the task of odometer type classification. Each model has its own strengths and weaknesses, which are essential to consider in the context of achieving optimal results.

MODEL	ACCURACY	PRECISION	RECALL	F1-SCORE
CNN	1.0	1.0	1.0	1.0
RANDOM FOREST	0.9791	0.9966	0.97	0.9831
KNN	0.90	0.87	0.99	0.92

*Table 1.6: Performance Comparison of three models on classification task.*

### 1.6.1 KNeighbors Classifier:

The KNeighbors Classifier initially demonstrated an accuracy of 86.61%, which was improved to 90% after hyperparameter tuning. The model relies on the majority class of k-nearest neighbors for classification, making it effective in capturing local patterns. However, its simplicity may limit its ability to discern complex relationships within the data. KNeighbors Classifier is computationally expensive, especially as the dataset grows, and may not perform well in high-dimensional spaces.

### 1.6.2 Random Forest Classifier:

The Random Forest Classifier exhibited robust performance with an accuracy of 97%. This ensemble method constructs multiple decision trees, mitigating overfitting and enhancing overall accuracy. Despite hyperparameter tuning not resulting in a significant accuracy boost, the model's ability to handle complex relationships in the data makes it a strong contender for odometer type classification. Random Forest Classifier is less sensitive to noisy data and performs well with a variety of features.

### 1.6.3 Convolutional Neural Network (CNN):

The CNN achieved nearly 100% accuracy on the training set, showcasing its ability to automatically learn hierarchical representations of features in image data. CNNs are well-suited for image-based tasks, making them ideal for odometer type classification. However, achieving high accuracy on the training set raises concerns about potential overfitting. CNNs require substantial computational resources for training and may not be the most efficient choice for simpler tasks.

## Comparison According to different aspects:

### Accuracy:

CNN outperformed both KNeighbors and Random Forest, achieving near-perfect accuracy on the training set. Random Forest performed exceptionally well with 97% accuracy, while KNeighbors lagged at 90%.

### Computational Efficiency:

KNeighbors Classifier is computationally expensive, especially with large datasets, whereas Random Forest and CNN may also demand significant computational resources but can handle larger and more complex datasets.

### Interpretability:

KNeighbors is straightforward to interpret, making it suitable for simpler tasks. Random Forest, despite being an ensemble, provides insight into feature importance. CNNs, on the other hand, are often considered as black-box models, making interpretation challenging.

In conclusion, the choice of the model depends on the specific requirements of the odometer type classification task. If interpretability and simplicity are paramount, KNeighbors might suffice. For robustness and high accuracy, Random Forest is a solid choice. If the task involves image data with intricate patterns, CNN stands out despite potential computational demands and concerns about overfitting. The final decision should consider the trade-offs between accuracy, interpretability, and computational efficiency based on the specific needs of the application.

## Task 2: Mileage Extraction

In mileage extraction part I use EasyOCR python library which has a pretrained model that using for extraction text from image. You can access this notebook by this [link](#).

Image preprocessing part:



*Figure 2a: From top to bottom: Cropped image of odometer, Gray scaled image, Thresh hold image.*

After these steps, I use EasyOCR extracting image from text. You can find the implementation on notebook.



Here is some of the results:

```
Actual Mileage: 244362, Predicted Mileage: 1244362, Accuracy: 16.67%
Actual Mileage: 64750, Predicted Mileage: 64750, Accuracy: 100.00%
Actual Mileage: 7863, Predicted Mileage: 78607, Accuracy: 75.00%
Actual Mileage: 18613, Predicted Mileage: No mileage detected., Accuracy: 0.00%
Actual Mileage: 35376, Predicted Mileage: 353, Accuracy: 60.00%
Actual Mileage: 159073, Predicted Mileage: 1590734, Accuracy: 100.00%
Actual Mileage: 244483, Predicted Mileage: 24448, Accuracy: 83.33%
Actual Mileage: 64750, Predicted Mileage: 647504, Accuracy: 100.00%
Actual Mileage: 4909, Predicted Mileage: 490, Accuracy: 75.00%
Actual Mileage: 2183, Predicted Mileage: 286, Accuracy: 25.00%Image: /content/drive/

Actual Mileage: 189350, Predicted Mileage: 109350, Accuracy: 83.33%
Actual Mileage: 479137, Predicted Mileage: 479136, Accuracy: 83.33%
Actual Mileage: 118316, Predicted Mileage: 1836, Accuracy: 16.67%
Actual Mileage: 148380, Predicted Mileage: No mileage detected., Accuracy: 0.00%
Actual Mileage: 3782, Predicted Mileage: 3814, Accuracy: 25.00%
Actual Mileage: 52672, Predicted Mileage: 52676, Accuracy: 80.00%
Actual Mileage: 115266, Predicted Mileage: 701100, Accuracy: 0.00%
Actual Mileage: 116950, Predicted Mileage: 116958, Accuracy: 83.33%
Actual Mileage: 26188, Predicted Mileage: 2608, Accuracy: 60.00%
Actual Mileage: 4909, Predicted Mileage: 490, Accuracy: 75.00%
Actual Mileage: 30721, Predicted Mileage: 3072, Accuracy: 80.00%
Actual Mileage: 26174, Predicted Mileage: No mileage detected., Accuracy: 0.00%
Actual Mileage: 115310, Predicted Mileage: 11530, Accuracy: 66.67%
```

*Figure 2b: Some of the outputs from mileage extraction.*

Since there are too many pictures, I didn't run the code for whole pictures. Also, I didn't use bounding boxes for digits because of the problem on bounding boxes.

Unfortunately, the accuracy results are not satisfactory at all.

### 3. Challenges and Potential Improvements

*To be completely honest, I must admit that I currently lack the necessary background to proficiently complete this assignment. Nevertheless, I am actively working on sharpening my skills in this area. Throughout the process, I conducted thorough research, learning a great deal. However, I hit some roadblocks in completing certain aspects, and I could not find satisfactory answers in some instances. In the second phase of the project, particularly in the "Mileage Extraction" part, I dedicated considerable time to training the model on dataset images and after testing it with the test set. Yet, this section turned out to be more challenging than expected, and I fell short in completing it. Additionally, I must mention that certain issues with annotations have hindered the completion of this study.*

*I am committed to furthering my knowledge and skills in this field and appreciate the opportunity to learn and progress through this assignment.*

*Discussions about problematic bounding boxes:*

*Most of the images have problematic bounding boxes. For example at Figure 3. There are unnecessary bounding boxes for date and time. Although, unnecessary ones are marked with x there are still digits in date and time. I tried to fix this issue but sadly, I didn't have enough time for that.*



*Figure 3: Sample image with problematic bounding boxes.*

## Conclusion

In this project, using Google Colab and Drive ensured data reliability for classifying odometer types and extracting mileage. The dataset, containing varied odometer images, underwent careful preprocessing, including cropping and normalization. Three models—KNeighbors, Random Forest, and CNN—were used, with CNN showing the best accuracy on the training set. While KNeighbors and Random Forest provided interpretability and efficiency, CNN excelled in tasks involving images. Mileage extraction, using EasyOCR, faced challenges with bounding boxes and achieved less satisfactory accuracy. Recognizing these challenges and expressing a commitment to continuous learning, the project highlighted the nuanced decision-making process in choosing models based on specific task requirements, emphasizing the iterative nature of problem-solving in machine learning.

## References

- [1] Yürekli, Ali; Yilmazel, Burcu; Mouheb, Kaouther (2021), "TRODOD: a public vehicle odometers dataset for computer vision", Mendeley Data, V1, doi: 10.17632/6y8m379mkt.1
- [2] Hjelm, M., & Andersson, E. (2022). Benchmarking Object Detection Algorithms for Optical Character Recognition of Odometer Mileage (Dissertation). Retrieved from <https://urn.kb.se/resolve?urn=urn:nbn:se:uu:diva-481952>