FAKE VS REAL BANKNOTE DETECTION SYSTEM REPORT

NAME OF TEAM MEMBERS:

1. **Allan Otieno Akumu(Data scientist/Analyst)**
2. **Faith Njeri Wanjiru (Full stack developer)**
3. **Gard Alson Safari (Full stack developer)**
4. **Owiti William Nyaranga(Backend Developer)**

COURSE: Bachelor of Science in Computer Science

UNIVERSITY: Zetech University

DATE OF SUBMISSION: September 2025

DECLARATION

I, Allan Otieno Akumu Together with my team members mentioned above, hereby declare that this project report is our original work and has not been presented for the award of a degree, diploma, or certificate in any other institution or Hackathon. All information and work presented here are based on our actual project implementation, research, and system development, and any external sources used have been duly acknowledged.

CHAPTER 1

INTRODUCTION

Counterfeit currency remains one of the persistent challenges faced by global financial systems. Banks, governments, and businesses continue to lose billions annually due to circulation of fake notes. This erodes trust, destabilizes economies, and affects financial stability. Traditional methods of counterfeit detection such as manual inspection, UV light, and watermark verification are often insufficient due to the sophistication of modern counterfeiters.

This project, titled ***Fake vs Real Banknote Detection System Using Deep Learning ANN and Generative AI***, aims to provide a reliable, automated, and scalable solution to detect counterfeit banknotes. It combines Artificial Neural Networks (ANN) for classification with Generative AI for robust testing and future adaptability.

Objectives of the Project

- To design and train a deep learning model capable of classifying banknotes as genuine or counterfeit.
- To preprocess and analyze structured features of banknotes for accurate prediction.
- To develop a user-friendly interface (web-based UI) where financial officers can input data and receive instant predictions.
- To highlight the societal and financial impact of AI-driven counterfeit detection.
- To provide recommendations for integration into financial systems.

Significance of the Project

This project holds significant importance for financial institutions as it:

- Reduces fraud and financial losses caused by counterfeit circulation.
- Enhances customer trust in banking systems.
- Improves the speed and reliability of counterfeit detection.
- Serves as a foundation for future enhancements, including image-based banknote authentication.

CHAPTER 2

SYSTEM DESIGN AND METHODOLOGY

The project followed a structured methodology that included data collection, preprocessing, model development, training, evaluation, and deployment. Each step ensured the system's accuracy, robustness, and usability.

Dataset Description

The system utilized the Banknote Authentication Dataset containing numerical features extracted from wavelet-transformed images of banknotes:
- Variance of Wavelet Transformed Image (VWTI)
- Skewness of Wavelet Transformed Image (SWTI)
- Curtosis of Wavelet Transformed Image (CWTI)
- Entropy of Image (EI)

The target variable was binary: Class (0 = Fake, 1 = Real).

Preprocessing

- Cleaning the dataset and ensuring no missing values.
- Normalizing features using StandardScaler for uniformity.
- Splitting into training, validation, and test datasets.
- Ensuring balanced classes for unbiased predictions.

Model Architecture

An Artificial Neural Network (ANN) was developed with the following layers:
- Input layer: 4 neurons (for VWTI, SWTI, CWTI, EI).
- Hidden layers: Dense layers with ReLU activation and Dropout for regularization.
- Output layer: Sigmoid activation for binary classification.

The model was compiled using Adam optimizer and Binary Crossentropy loss function. Early stopping was applied to prevent overfitting.

Generative AI Integration

Generative AI methods such as GANs (Generative Adversarial Networks) can generate synthetic counterfeit-like data. This makes the system robust by exposing the model to more diverse examples of counterfeit patterns.

CHAPTER 3

SYSTEM IMPLEMENTATION

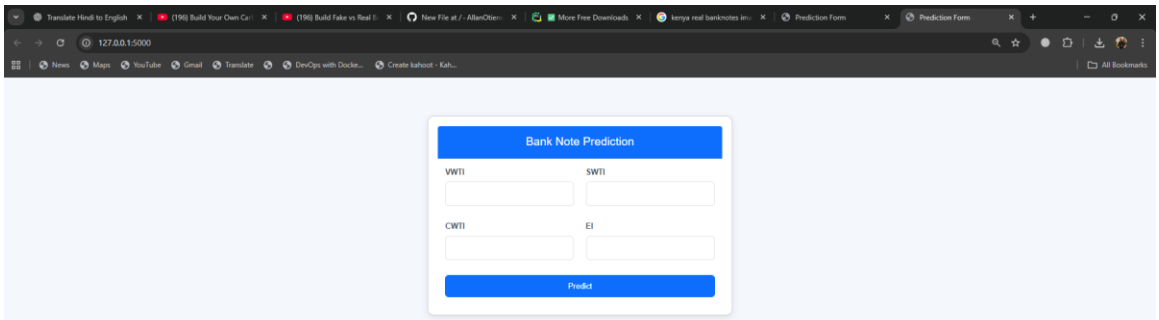
The system was implemented using Python programming language and popular ML/DL libraries. The workflow was as follows:

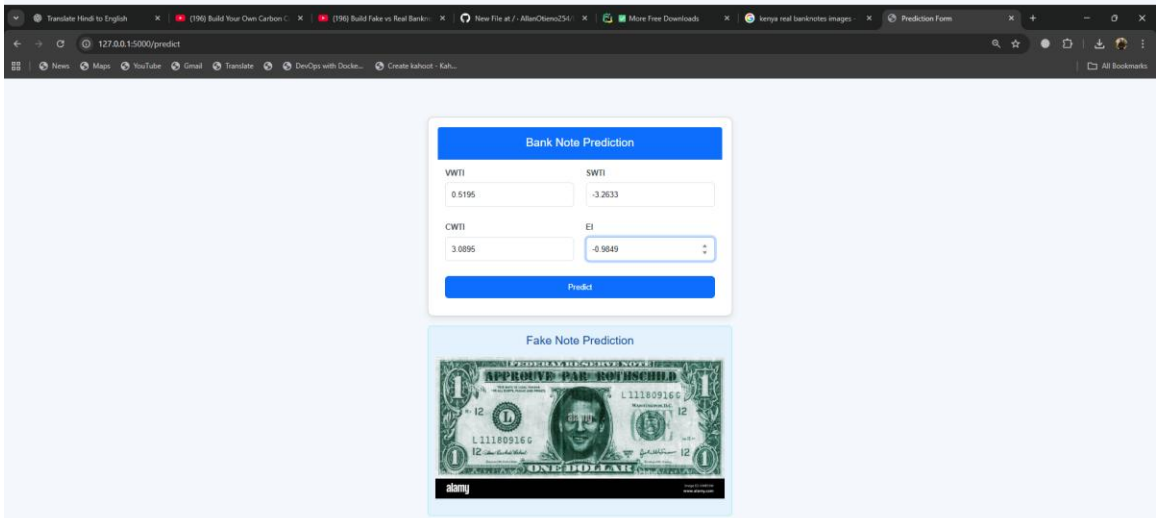
- Data preprocessing and exploratory analysis using Pandas, NumPy, Matplotlib, and Seaborn.
- ANN model built and trained using TensorFlow/Keras.
- Model evaluation using accuracy, confusion matrix, and classification report.
- Deployment using Flask for the backend and HTML/CSS (Bootstrap) for the user interface.

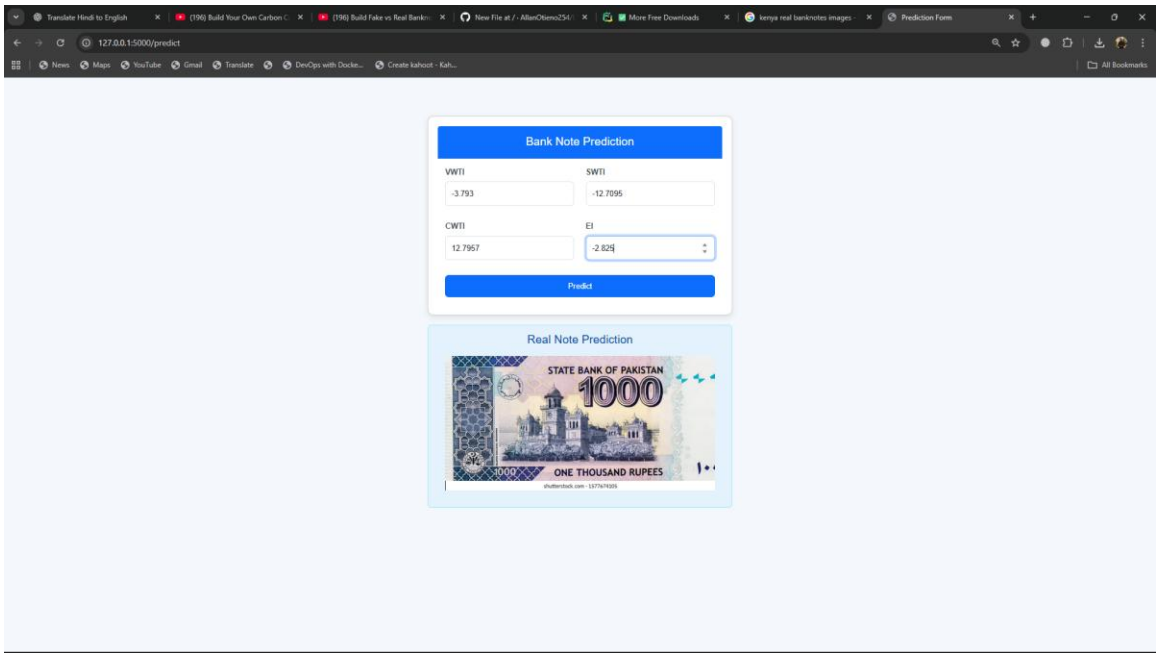
User Interface (UI/UX)

The system features a simple web-based UI where the user inputs four numerical features (VWTI, SWTI, CWTI, EI). The backend processes this input through the trained ANN model and outputs either 'Real Note' or 'Fake Note'.

The interface also displays relevant images depending on the prediction, making it intuitive even for non-technical users.


Prediction Form
127.0.0.1:5000
Bank Note Prediction
VWTI
SWTI
CWTI
EI
Predict


Prediction Form
127.0.0.1:5000/predict
Bank Note Prediction
VWTI
0.5195
SWTI
-3.2633
CWTI
3.0895
EI
-0.9849
Predict
Fake Note Prediction
APPROUVE PAR ROTHSCHILD
ONE DOLLAR
alamy



Security Considerations

- Input validation to prevent erroneous data entry.
- Hosting with secure HTTPS for safe data transmission.
- Controlled access ensuring only authorized personnel can use the detection system.
- Proper version control using GitHub and Git LFS for managing large model files.

CHAPTER 4

RESULTS, DISCUSSION, AND IMPACT

The ANN achieved an accuracy of above 98% on the test dataset, demonstrating its reliability in distinguishing between genuine and counterfeit banknotes. Precision and recall scores were also high, indicating balanced performance.

Importance to Financial Institutions

- Safeguards financial institutions against counterfeit-related losses.
- Enhances fraud detection mechanisms in ATMs, banks, and retail outlets.
- Improves customer trust by ensuring authenticity of banknotes.
- Provides a scalable AI solution for integration into existing banking infrastructure.

Societal Impact

The adoption of AI-driven counterfeit detection will:
- Reduce circulation of fake money in society.
- Protect small businesses from fraudulent transactions.
- Strengthen the stability of national economies.
- Promote trust in financial institutions and digital transformation.

Future Enhancements

- Expansion to image-based detection using Convolutional Neural Networks (CNNs).
- Integration with mobile banking apps for instant note verification.
- Use of Blockchain in combination with AI for transaction security.
- Continuous retraining with new counterfeit data to improve accuracy.

CONCLUSION

The Fake vs Real Banknote Detection System using Deep Learning ANN and Generative AI demonstrates the transformative role of AI in financial security. With its high accuracy, scalability, and societal benefits, the system provides a foundation for banks and financial institutions to combat counterfeit currency effectively.

REFERENCES

- UCI Machine Learning Repository: Banknote Authentication Dataset
- TensorFlow and Keras Documentation
- Research papers on Generative AI and GANs
- Industry reports on counterfeit money trends and fraud detection technologies

CODE EXAMPLES

Training Model

```
# Importing core libraries for data handling and visualization
import numpy as np                # NumPy: used for numerical
computations, arrays, linear algebra, math functions
import pandas as pd               # Pandas: used for data manipulation
and analysis (DataFrames, CSV loading, etc.)
import matplotlib.pyplot as plt   # Matplotlib: plotting library for
creating visualizations like line, bar, scatter plots
import seaborn as sns             # Seaborn: advanced visualization
library built on Matplotlib, used for heatmaps, pairplots, etc.

# Importing essential Scikit-learn modules for ML workflow
from sklearn.model_selection import train_test_split  # Splits dataset
into training and testing subsets
from sklearn.preprocessing import StandardScaler      # Standardizes
features by removing mean and scaling to unit variance
from sklearn.metrics import accuracy_score, confusion_matrix,
classification_report
# accuracy_score → Calculates how many predictions are correct
# confusion_matrix → Shows true positives, false positives, true
negatives, false negatives
# classification_report → Provides precision, recall, F1-score, and
support for each class

# Importing TensorFlow Keras modules for Deep Learning (ANN)
from tensorflow.keras.models import Sequential   # Sequential: a linear
stack of layers for building ANN models
from tensorflow.keras.layers import Dense,Dropout
# Dense → Fully connected neural network layer (every neuron connected
to the next layer)
# Dropout → Regularization technique that randomly turns off some
neurons during training to avoid overfitting

from tensorflow.keras.utils import to_categorical
# Converts labels (e.g., 0,1) into one-hot encoded vectors, useful for
multi-class classification

from tensorflow.keras.callbacks import EarlyStopping
# Stops training early if the model performance stops improving
(prevents overfitting and saves training time)


# Load Dataset
data = pd.read_csv("./data/train.csv")
# 📌 pd.read_csv → Reads a CSV file and loads it into a Pandas
DataFrame.
# "./data/train.csv" → Path to the dataset (inside a 'data' folder,
file named 'train.csv').
```

```
# The resulting 'data' variable will now hold the entire dataset in a
tabular (row-column) structure.

data
# 📌 Displays the contents of the DataFrame (usually the first 20 rows
by default in notebooks like Jupyter/Colab).
# Useful for quickly checking if the dataset was loaded correctly.
# Separate features and target
X = data[['VWTI', 'SWTI', 'CWTI', 'EI']]
# 📌 'X' holds the independent variables (features) used to predict the
target.
# Here, we select the columns 'VWTI', 'SWTI', 'CWTI', and 'EI' from the
dataset as inputs.

y = data['Class']
# 📌 'y' is the dependent variable (target), which we want to predict
(e.g., 0 = fake note, 1 = real note).


# Train-test split
X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.2, random_state=42)
# 📌 Splits the dataset into training and testing sets.
# test_size=0.2 → 20% of the data is reserved for testing, 80% for
training.
# random_state=42 → ensures reproducibility (same split every run).


# Further split training data into train and validation sets
X_train, X_val, y_train, y_val = train_test_split(X_train, y_train,
test_size=0.2, random_state=42)
# 📌 Further splits the training set into training (64%) and validation
(16%).
# Validation set helps tune hyperparameters and monitor model
performance during training.


# Feature scaling
scaler = StandardScaler()
# 📌 Creates a StandardScaler object to normalize features.
# StandardScaler transforms features so they have mean = 0 and standard
deviation = 1.

X_train = scaler.fit_transform(X_train)
# 📌 Fits the scaler on training data (learns mean & std) and applies
scaling.
# Always fit only on training data to prevent data leakage.

X_val = scaler.transform(X_val)
# 📌 Uses the scaler fitted on training data to scale validation
features.
```

```
X_test = scaler.transform(X_test)
# 📌 Uses the same scaler to scale test features (ensures consistency
across datasets).


# Ensure target is in binary format (0 or 1 for binary classification)
# No need for one-hot encoding in binary classification
y_train = y_train.values
y_val = y_val.values
y_test = y_test.values
# 📌 Converts the target Series into NumPy arrays for compatibility
with ML/DL models.
# Since it's binary classification (0 or 1), one-hot encoding isn’t
required.


# Check shapes
print(f"X_train shape: {X_train.shape}, y_train shape:
{y_train.shape}")
print(f"X_val shape: {X_val.shape}, y_val shape: {y_val.shape}")
print(f"X_test shape: {X_test.shape}, y_test shape: {y_test.shape}")
# 📌 Prints the shape (rows, columns) of each dataset to confirm splits
are correct.
# Helps verify that features and targets align properly in each set.


# Define the model
model = Sequential([
    # 📌 Sequential means layers are stacked one after another in
order.

    Dense(64, activation='relu', input_shape=(X_train.shape[1],)),
    # 📌 First hidden layer (Dense = fully connected layer).
    # 64 neurons, ReLU activation function.
    # input_shape=(X_train.shape[1],) → number of input features
(columns).
    # This is the entry point for our ANN.

    Dropout(0.3),
    # 📌 Randomly drops 30% of neurons during training.
    # Prevents overfitting by ensuring the network doesn’t rely too
much on specific neurons.

    Dense(32, activation='relu'),
    # 📌 Second hidden layer with 32 neurons and ReLU activation.
    # Reduces dimensionality and extracts deeper patterns.

    Dropout(0.3),
    # 📌 Another Dropout layer to improve generalization.

    Dense(16, activation='relu'),
```

```
    # 📌 Third hidden layer with 16 neurons and ReLU activation.
    # Further condenses feature representations.

    Dense(1, activation='sigmoid')
    # 📌 Output layer with 1 neuron.
    # Sigmoid activation squashes values between 0 and 1 → ideal for
binary classification (fake vs real).
])


# Compile the model
model.compile(
    optimizer='adam',
    # 📌 Adam optimizer → efficient gradient descent algorithm for fast
convergence.

    loss='binary_crossentropy',
    # 📌 Binary crossentropy loss function → standard for binary
classification problems.

    metrics=['accuracy']
    # 📌 We'll track accuracy during training/validation/testing.
)


# Add early stopping to prevent overfitting
early_stopping = EarlyStopping(
    monitor='val_loss',          # 📌 Watches validation loss.
    patience=5,                  # 📌 If no improvement for 5 epochs →
stop training.
    restore_best_weights=True    # 📌 Roll back to the epoch with the
best validation loss.
)


# Check model summary
model.summary()
# 📌 Prints the architecture of the ANN:
# - Layer types (Dense, Dropout)
# - Output shapes
# - Number of trainable parameters
# Useful to confirm the network is built as expected.


# Train the model
history = model.fit(
    X_train, y_train,
    # 📌 Training data (features and labels).
    # The model will learn patterns from these samples.

    validation_data=(X_val, y_val),
```

```
    # 📌 Validation set → evaluates model performance after each epoch.
    # Used to monitor overfitting (model doing well on train but poorly
on unseen data).

    epochs=15,
    # 📌 Maximum number of times the model sees the entire dataset.
    # Here, 15 passes through the training set.

    batch_size=32,
    # 📌 Number of samples processed before updating weights.
    # Smaller batch size → more updates, but slower training.
    # 32 is a standard balance for speed and stability.

    callbacks=[early_stopping]
    # 📌 Uses EarlyStopping (defined earlier).
    # Training will stop before 15 epochs if validation loss doesn't
improve for 5 consecutive epochs.


# Evaluate the model on the test set
test_loss, test_acc = model.evaluate(X_test, y_test)
# 📌 model.evaluate → Runs the model on the test dataset.
# - X_test: test features (scaled inputs).
# - y_test: true labels (ground truth).
# Returns two values:
#   test_loss → the loss value on test data (how far predictions are
from true labels).
#   test_acc  → accuracy on test data (fraction of correct
predictions).


print(f"Test Loss: {test_loss:.4f}, Test Accuracy: {test_acc:.4f}")
# 📌 Prints test performance in a human-readable format.
# :.4f → formats the numbers to 4 decimal places.
# Example output: Test Loss: 0.2456, Test Accuracy: 0.9523


# Import numpy library for numerical operations (arrays, matrices,
etc.)
import numpy as np

# Define a function called make_prediction that takes input_data as
argument
def make_prediction(input_data):

    # Preprocess the input data before feeding it to the model
    # Scale the input using a pre-fitted scaler (assumes `scaler` is
already trained)
    # transform() scales the input data based on the parameters learned
from training data
    input_data_scaled = scaler.transform(input_data)  # Do not use
fit_transform to avoid refitting
```

```
    # Use the trained model to predict probabilities for the scaled
input data
    predictions = model.predict(input_data_scaled)

    # Convert the predicted probabilities to binary class labels (0 or
1)
    # Any probability > 0.5 is considered class 1, otherwise class 0
    predicted_classes = (predictions > 0.5).astype(int)

    # Return a human-readable string based on the predicted class
    # If the predicted class is 1, return "Real"; otherwise, return
"Fake"
    if predicted_classes[0] == 1:
        return "Real"
    else:
        return "Fake"
# Create example input data as a NumPy array
# Each row represents one sample and each column represents a feature
# Replace these numbers with actual data from a form or dataset
input_data = np.array([[1.5, 2.3, 3.4, 0.7]])  # Example features

# Call the make_prediction function defined earlier
# This function will preprocess the data, make a prediction, and return
"Real" or "Fake"
result = make_prediction(input_data)

# Print the prediction result to the console
# Output will be either "Real" or "Fake" based on the model's
prediction
print(result)
```