Chapter 3

Methodology and Implementation

3.1 Block diagram

3.1.1 Activity diagram

The activity diagram Fig 3.1.1 outlines the process of signal classification. It starts with loading the data, followed by adding noise to simulate real-world conditions and labeling the data accordingly. The dataset is then split into training and testing sets. Three machine learning models—SVM, Random Forest (RF), and Convolutional Neural Networks (CNN)—are trained on the training data. After training, predictions are made on the testing data, and the models' performance is evaluated using accuracy metrics, confusion matrices, and the ROC curve to assess classification effectiveness.

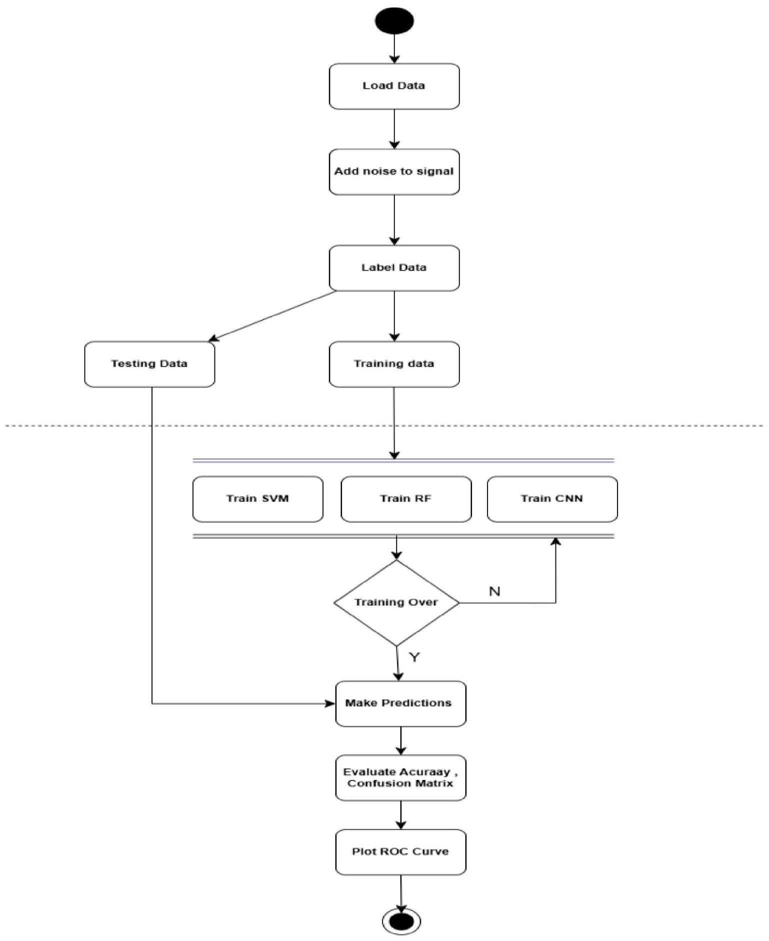


Fig 3.1.1 Activity Diagram

3.1.2 Data flow diagram

The Data Flow Diagram - Level 0 Fig 3.1.2. shows the interaction between the user and the system, where the user inputs signal data, and the system classifies and returns information about the signal.

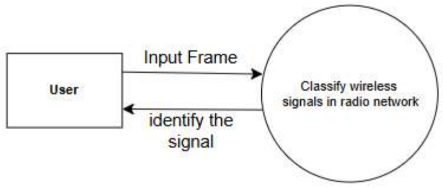


Fig 3.1.2 Data Flow Diagram - Level 0

3.1.3 Sequence diagram

This sequence diagram Fig 3.1.3 illustrates the process of classifying wireless signals by interacting between a Signal Processor, Data Manager, and Machine Learning Models. It starts with the signal processor loading the signal data, followed by the data manager adding noise to the signals and labeling them as either signal with noise or noise-only. The labeled dataset is then split into training and testing sets. Next, machine learning algorithms like SVM and Random Forest are applied, followed by the application of a CNN model. The results from both models are returned to the data manager and then to the signal processor. Finally, the signal processor automates modulation type detection, optimizes spectrum use, and improves the accuracy and efficiency of signal classification.

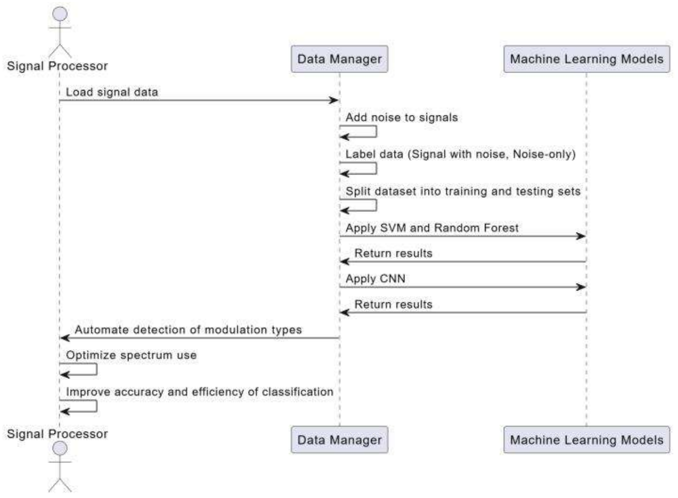

Signal Processor
Data Manager
Machine Learning Models
Load signal data
Add noise to signals
Label data (Signal with noise, Noise-only)
Split dataset into training and testing sets
Apply SVM and Random Forest
Return results
Apply CNN
Return results
Automate detection of modulation types
Optimize spectrum use
Improve accuracy and efficiency of classification
Signal Processor
Data Manager
Machine Learning Models

Fig 3.1.3 Sequence Diagram

3.2 System Design and Architecture

3.2.1 Data Processing

The dataset [22] for spectrum sensing is processed by adding Gaussian noise to signals and preparing it for classification. It starts by generating Gaussian noise using the function **gaussian_noise(mu, sigma)**, which produces a 2x128 matrix of noise with a mean of 0 and a standard deviation that matches the signal. This noise is then added to the original signal through the **noise_added(signal, Gnoise)** function, simulating a real-world noisy environment. The standard deviation of each signal is calculated using **sd_calc(data)** to ensure the noise intensity aligns with the signal's characteristics. For each signal in the dataset, the code computes its standard deviation, generates corresponding noise, adds it to the signal, and stores both the noisy signal and the noise itself.

The noisy signals are stored in a new dataset, and the original dataset is updated with this noise-added data. Labels are assigned to the data: `1` for signal-plus-noise and `0` for pure noise, enabling the model to distinguish between the two during classification. Both datasets (signal-plus-noise and pure noise) are concatenated and shuffled to randomize the order for unbiased training as shown in Fig.3.2.1. The final dataset includes columns for modulation type, SNR, the signal data (with or without noise), and labels. This processed dataset is then used to train a machine learning model to detect signals in noisy environments, an essential task in spectrum sensing, especially in applications like cognitive radios, where distinguishing between noise and legitimate signals is crucial for efficient spectrum utilization.

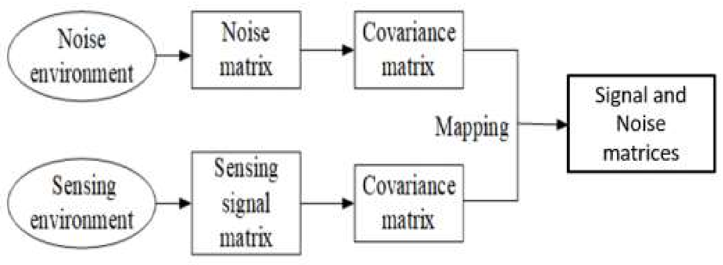


Fig. 3.2.1 Data Formation

3.2.2 Convolution Neural Network Architecture

The flowchart outlines a structured approach to spectrum sensing using a Convolutional Neural Network (CNN) in Fig 3.2.2.1. It begins with data preparation, which involves feature extraction and data preprocessing. During this phase, the training data is directed towards the model for further processing, while the test data is set aside for later evaluation. Next, the spectrum sensing stage commences with the initialization of the CNN model.

The model then processes the training data to learn the relevant features. After this step, the process checks whether the training is complete. If the training is ongoing, it loops back to refine the model further. Once the training is finished, the CNN is ready to perform spectrum sensing on the test data. This methodical workflow ensures that the model is thoroughly trained before applying it to detect signals amidst noise, ultimately improving the accuracy and efficiency of spectrum sensing.

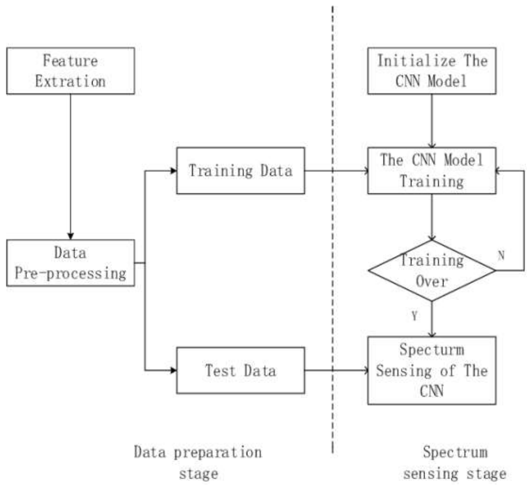
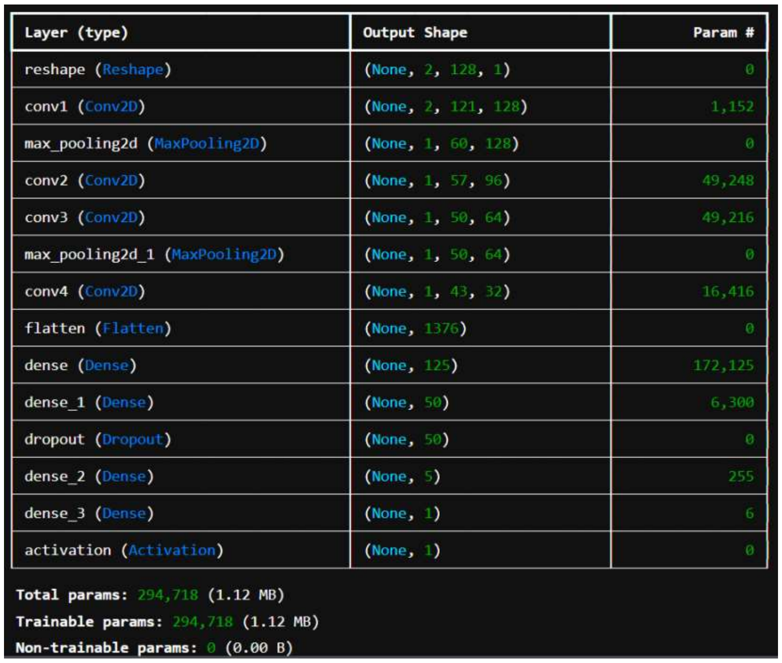


Fig 3.2.2.1 CNN Flowchart

The Convolutional Neural Network (CNN) using Keras for a binary classification task, likely related to spectrum sensing. The architecture begins with a reshaping layer that converts the input shape into a 2D format suitable for convolutional processing. It consists of several convolutional layers (Conv2D), each followed by activation functions (ReLU) to introduce non-linearity as shown in Fig 3.2.2.3. The network employs MaxPooling2D layers to down sample the feature maps, reducing spatial dimensions and computational load while retaining important features. After the convolutional layers, the output is flattened into a one-dimensional array, which is then fed into fully connected (Dense) layers.

The final layer uses a sigmoid activation function to output a probability score for binary classification. The model is compiled with a binary cross-entropy loss function and the Adam optimizer, making it suitable for training on noisy signal data to distinguish between signal presence and absence. The summary of the model indicates a total of 294,718 parameters, all of which are trainable, showcasing the complexity of the architecture designed to capture the underlying patterns in the data.

In this project, a Convolutional Neural Network (CNN) is chosen for its effectiveness in analyzing complex patterns in wireless signal data, essential for accurately classifying modulation types. CNNs automatically extract relevant features from signals, handle large datasets efficiently through parameter sharing and dimensionality reduction, and are well-suited for two-dimensional representations like spectrograms.



Layer (type)	Output Shape	Param #
reshape (Reshape)	(None, 2, 128, 1)	0
conv1 (Conv2D)	(None, 2, 121, 128)	1,152
max_pooling2d (MaxPooling2D)	(None, 1, 60, 128)	0
conv2 (Conv2D)	(None, 1, 57, 96)	49,248
conv3 (Conv2D)	(None, 1, 50, 64)	49,216
max_pooling2d_1 (MaxPooling2D)	(None, 1, 50, 64)	0
conv4 (Conv2D)	(None, 1, 43, 32)	16,416
flatten (Flatten)	(None, 1376)	0
dense (Dense)	(None, 125)	172,125
dense_1 (Dense)	(None, 50)	6,300
dropout (Dropout)	(None, 50)	0
dense_2 (Dense)	(None, 5)	255
dense_3 (Dense)	(None, 1)	6
activation (Activation)	(None, 1)	0

```
Total params: 294,718 (1.12 MB)
Trainable params: 294,718 (1.12 MB)
Non-trainable params: 0 (0.00 B)
```

Fig . 3.2.2.2. CNN Architecture used in the project.

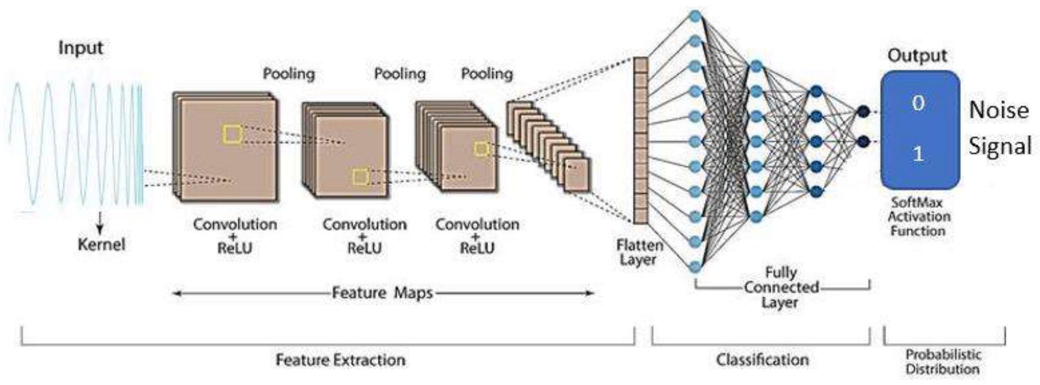


Fig . 3.2.2.3. CNN Architecture

3.2.3 Support Vector Machine and Random Forest

The flowchart outlines a machine learning workflow in Fig.3.2.3. It begins with loading the data, followed by splitting the data into training and testing sets. The training data is then used to train two models: Support Vector Machine (SVM) and Random Forest (RF). The training process continues iteratively until it meets a stopping condition. Once the training is complete, predictions are made using the trained models, and the results are evaluated for accuracy using the testing data.

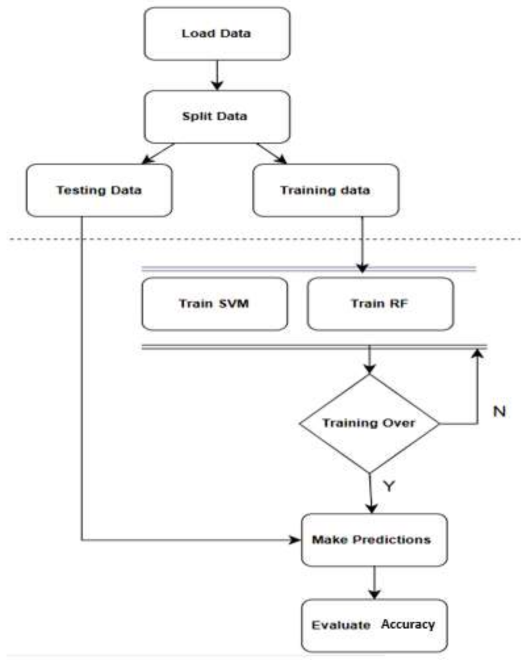


Fig . 3.2.3.SVM and RF Flowchart

3.3 Implementation and Testing

3.3.1 System Requirements

3.3.1.1 Hardware

- Windows 8 or above/ Linux
- RAM > 6 Gb
- Processor Intel i5 or above

3.3.1.2 Software

- Python 2.7.6 or above
- Jupyter Notebook/ Google Colab

3.3.2 Code Implementation

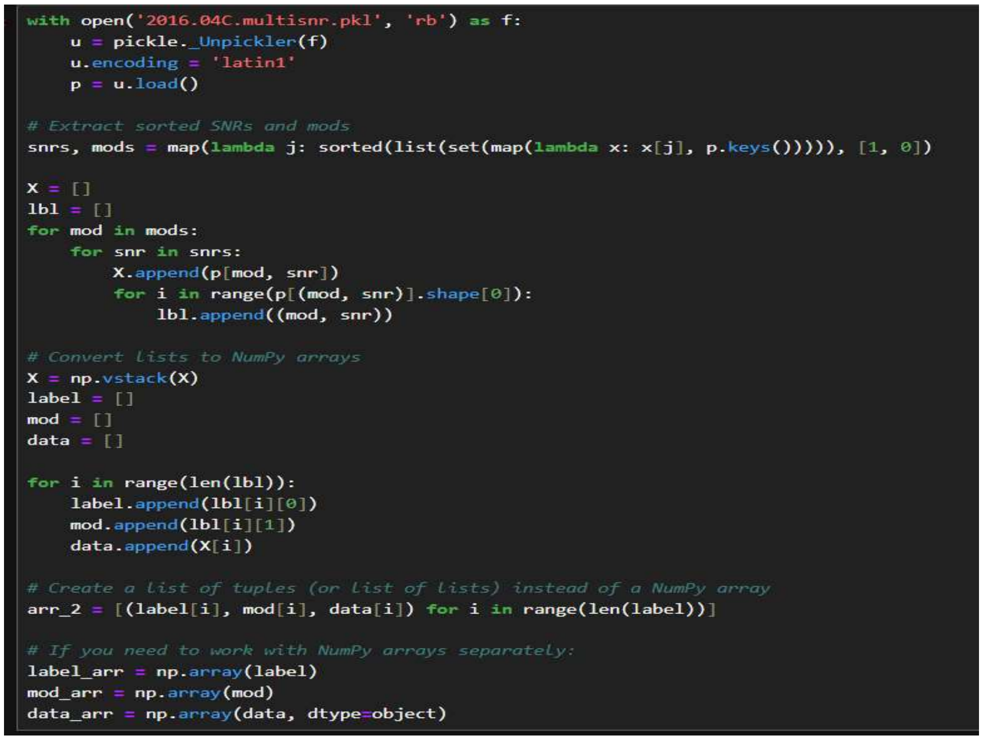
The data we have is in the pickle form which consists of 11 modulations with their respective SNR and IQ values. The data is processed by creating a labeled dataset by adding Gaussian noise to signals and preparing it for use in training a classifier for spectrum sensing, where the classifier's goal is to detect signals in noisy environments, which distinguishes between whether the primary user signal is present or absent, i.e., a binary classification. The entire dataset contains 161800 rows and 4 columns, (161800,4). The data set is then typically split into two parts i.e., training (70%)and testing datasets(30%).

Followed by Support Vector Machine (SVM) and Random Forest (RF) classifiers are implemented to classify signals after data preprocessing. The training and test data are first flattened and standardized using `StandardScaler`. The SVM model is initialized with a linear kernel and a maximum of 1000 iterations, while the RF model uses 100 trees with a maximum depth of 10. Both models are trained on the scaled training data and then used to predict labels for the test set. The accuracy of each model is calculated using `accuracy_score` and printed, showing the performance of the SVM and RF classifiers

.

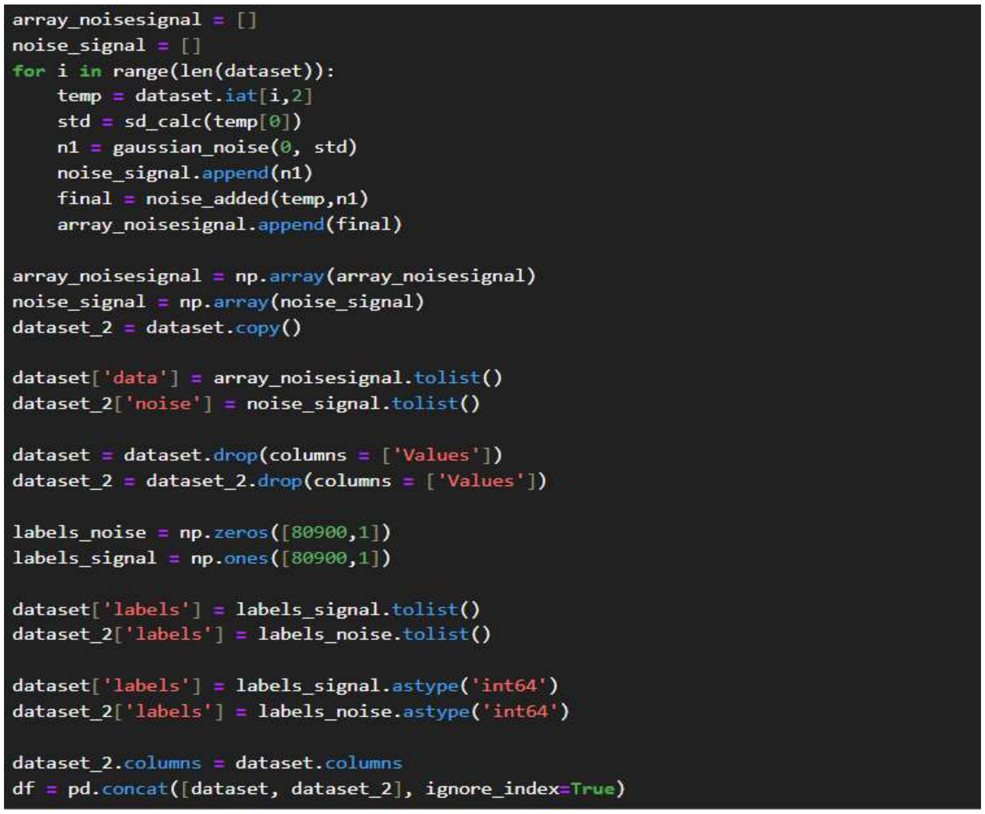
Next, the CNN model pipelines then train and test on the dataset giving us the accuracy of the classification as the output as the end result. The accuracy achieved is using Confusion matrix as performance measure of the model’s acurracy and plotting ROC curve.

3.3.2.1 Data Loading



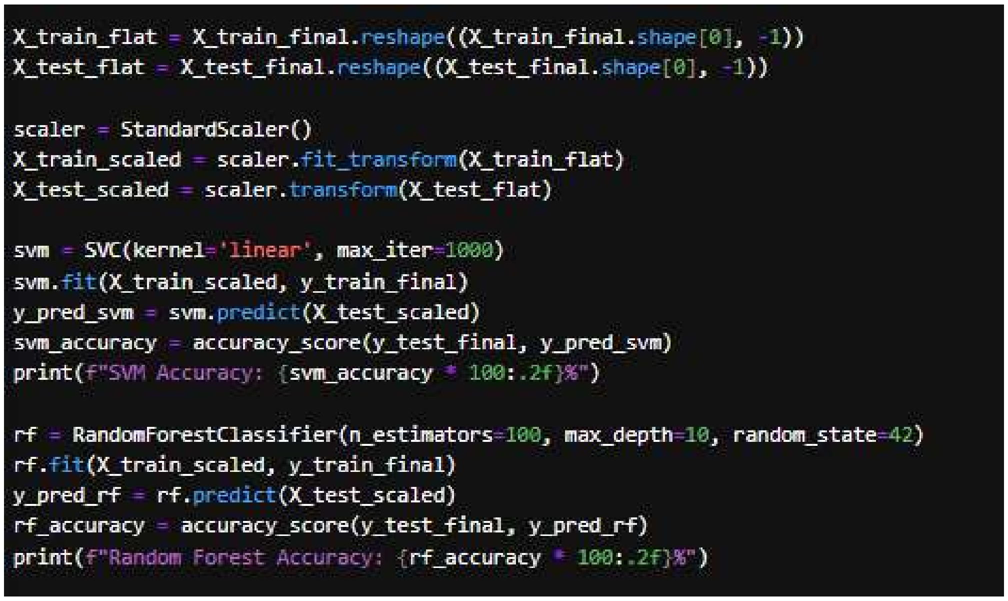
```
with open('2016.04C.multisnr.pkl', 'rb') as f:
    u = pickle._Unpickler(f)
    u.encoding = 'latin1'
    p = u.load()

# Extract sorted SNRs and mods
snrs, mods = map(lambda j: sorted(list(set(map(lambda x: x[j], p.keys())))), [1, 0])

X = []
lbl = []
for mod in mods:
    for snr in snrs:
        X.append(p[mod, snr])
        for i in range(p[(mod, snr)].shape[0]):
            lbl.append((mod, snr))

# Convert lists to NumPy arrays
X = np.vstack(X)
label = []
mod = []
data = []

for i in range(len(lbl)):
    label.append(lbl[i][0])
    mod.append(lbl[i][1])
    data.append(X[i])

# Create a list of tuples (or list of lists) instead of a NumPy array
arr_2 = [(label[i], mod[i], data[i]) for i in range(len(label))]

# If you need to work with NumPy arrays separately:
label_arr = np.array(label)
mod_arr = np.array(mod)
data_arr = np.array(data, dtype=object)
```

3.3.2.2 Data Processing



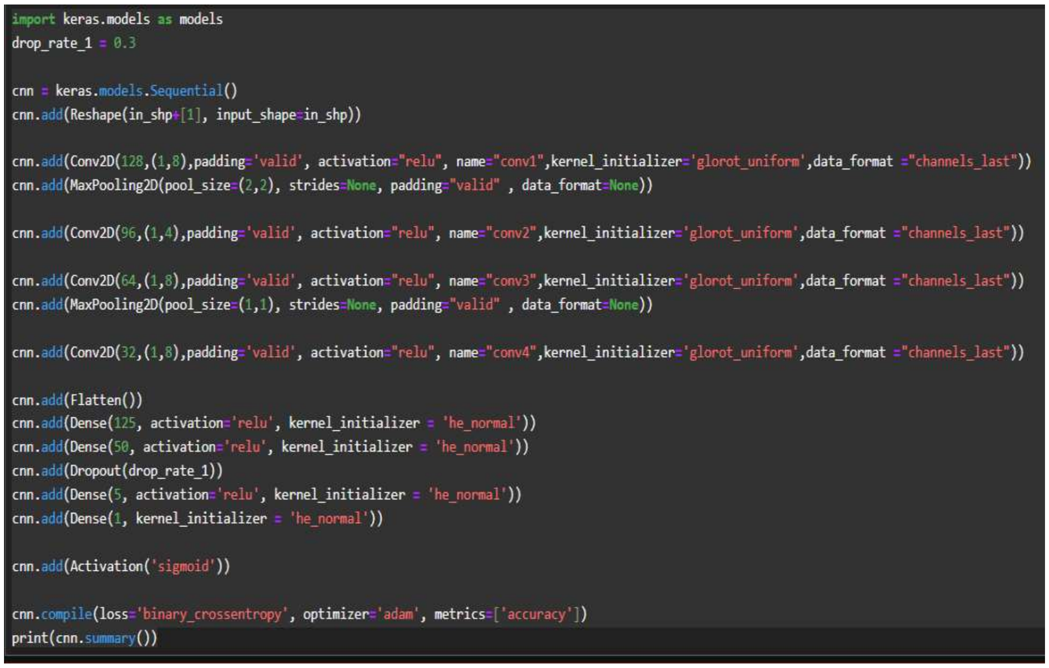
```
array_noisesignal = []
noise_signal = []
for i in range(len(dataset)):
    temp = dataset.iat[i,2]
    std = sd_calc(temp[0])
    n1 = gaussian_noise(0, std)
    noise_signal.append(n1)
    final = noise_added(temp,n1)
    array_noisesignal.append(final)

array_noisesignal = np.array(array_noisesignal)
noise_signal = np.array(noise_signal)
dataset_2 = dataset.copy()

dataset['data'] = array_noisesignal.tolist()
dataset_2['noise'] = noise_signal.tolist()

dataset = dataset.drop(columns = ['Values'])
dataset_2 = dataset_2.drop(columns = ['Values'])

labels_noise = np.zeros([80900,1])
labels_signal = np.ones([80900,1])

dataset['labels'] = labels_signal.tolist()
dataset_2['labels'] = labels_noise.tolist()

dataset['labels'] = labels_signal.astype('int64')
dataset_2['labels'] = labels_noise.astype('int64')

dataset_2.columns = dataset.columns
df = pd.concat([dataset, dataset_2], ignore_index=True)
```

3.3.2.3 SVM and RF implementation



```
X_train_flat = X_train_final.reshape((X_train_final.shape[0], -1))
X_test_flat = X_test_final.reshape((X_test_final.shape[0], -1))

scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train_flat)
X_test_scaled = scaler.transform(X_test_flat)

svm = SVC(kernel='linear', max_iter=1000)
svm.fit(X_train_scaled, y_train_final)
y_pred_svm = svm.predict(X_test_scaled)
svm_accuracy = accuracy_score(y_test_final, y_pred_svm)
print(f"SVM Accuracy: {svm_accuracy * 100:.2f}%")

rf = RandomForestClassifier(n_estimators=100, max_depth=10, random_state=42)
rf.fit(X_train_scaled, y_train_final)
y_pred_rf = rf.predict(X_test_scaled)
rf_accuracy = accuracy_score(y_test_final, y_pred_rf)
print(f"Random Forest Accuracy: {rf_accuracy * 100:.2f}%")
```

3.3.2.3 Building Model for CNN



```
import keras.models as models
drop_rate_1 = 0.3

cnn = keras.models.Sequential()
cnn.add(Reshape(in_shp+[1], input_shape=in_shp))

cnn.add(Conv2D(128,(1,8),padding='valid', activation="relu", name="conv1",kernel_initializer='glorot_uniform',data_format ="channels_last"))
cnn.add(MaxPooling2D(pool_size=(2,2), strides=None, padding="valid" , data_format=None))

cnn.add(Conv2D(96,(1,4),padding='valid', activation="relu", name="conv2",kernel_initializer='glorot_uniform',data_format ="channels_last"))

cnn.add(Conv2D(64,(1,8),padding='valid', activation="relu", name="conv3",kernel_initializer='glorot_uniform',data_format ="channels_last"))
cnn.add(MaxPooling2D(pool_size=(1,1), strides=None, padding="valid" , data_format=None))

cnn.add(Conv2D(32,(1,8),padding='valid', activation="relu", name="conv4",kernel_initializer='glorot_uniform',data_format ="channels_last"))

cnn.add(Flatten())
cnn.add(Dense(125, activation='relu', kernel_initializer = 'he_normal'))
cnn.add(Dense(50, activation='relu', kernel_initializer = 'he_normal'))
cnn.add(Dropout(drop_rate_1))
cnn.add(Dense(5, activation='relu', kernel_initializer = 'he_normal'))
cnn.add(Dense(1, kernel_initializer = 'he_normal'))

cnn.add(Activation('sigmoid'))

cnn.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
print(cnn.summary())
```

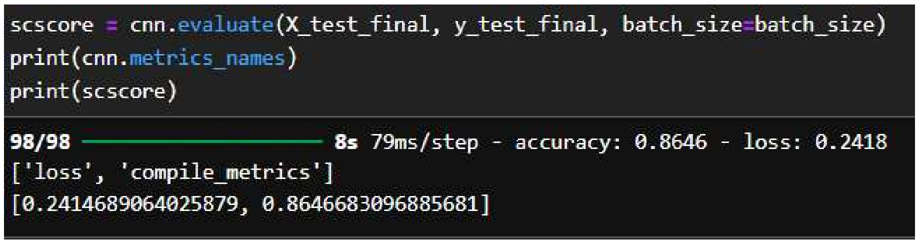
Chapter 4

Results and Analysis

4.1. Results

4.1.1 Accuracy of Model

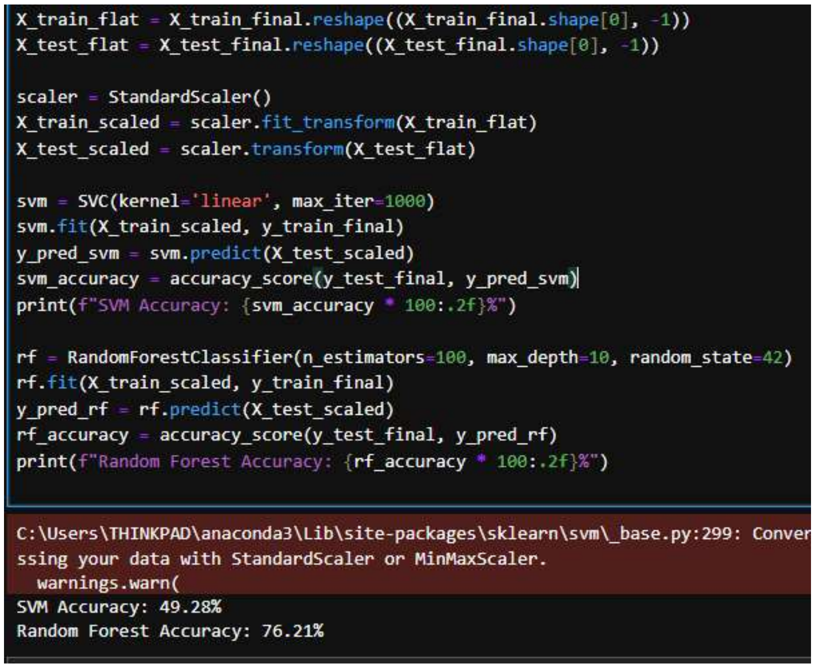
The outcome of applying the CNN algorithm, when the input signal shows whether a primary user is present in the radio environment, achieved an accuracy of 86%. This accuracy was assessed using test data, focusing on the model's ability to accurately predict the modulation types of the signals. The effectiveness of the model in distinguishing between various signal types (influenced by features such as SNR) during training plays a crucial role in this accuracy. In contrast, the SVM and RF methods yielded accuracies of 49.28% and 76.21%, respectively.



```
scscore = cnn.evaluate(X_test_final, y_test_final, batch_size=batch_size)
print(cnn.metrics_names)
print(scscore)
```

```
98/98 ━━━━━━━━━━━━━━━━━━━━ 8s 79ms/step - accuracy: 0.8646 - loss: 0.2418
['loss', 'compile_metrics']
[0.2414689064025879, 0.8646683096885681]
```

Fig. 4.1.1.1 CNN Model Accuracy



```
X_train_flat = X_train_final.reshape((X_train_final.shape[0], -1))
X_test_flat = X_test_final.reshape((X_test_final.shape[0], -1))

scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train_flat)
X_test_scaled = scaler.transform(X_test_flat)

svm = SVC(kernel='linear', max_iter=1000)
svm.fit(X_train_scaled, y_train_final)
y_pred_svm = svm.predict(X_test_scaled)
svm_accuracy = accuracy_score(y_test_final, y_pred_svm)
print(f"SVM Accuracy: {svm_accuracy * 100:.2f}%")

rf = RandomForestClassifier(n_estimators=100, max_depth=10, random_state=42)
rf.fit(X_train_scaled, y_train_final)
y_pred_rf = rf.predict(X_test_scaled)
rf_accuracy = accuracy_score(y_test_final, y_pred_rf)
print(f"Random Forest Accuracy: {rf_accuracy * 100:.2f}%")
```

```
C:\Users\THINKPAD\anaconda3\Lib\site-packages\sklearn\svm\_base.py:299: Conver
ssing your data with StandardScaler or MinMaxScaler.
  warnings.warn(
SVM Accuracy: 49.28%
Random Forest Accuracy: 76.21%
```

Fig 4.1.1.2 SVM and RF Accuracy

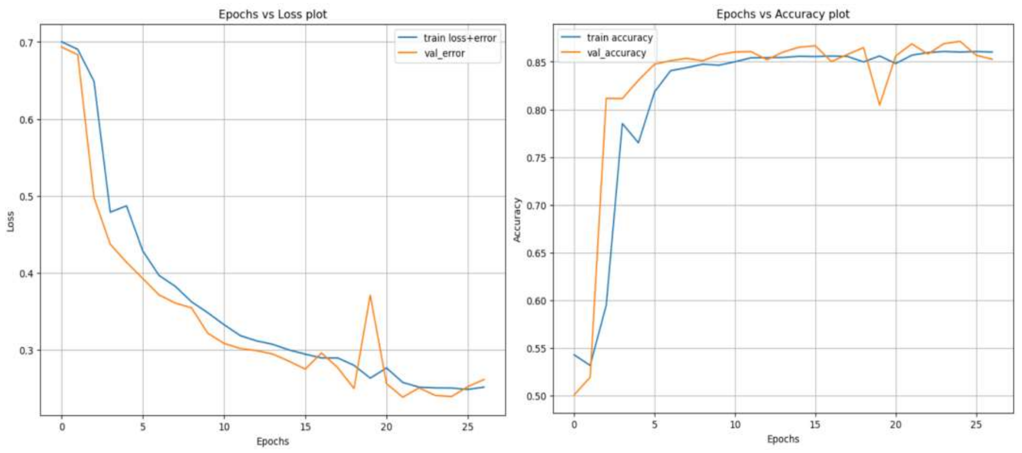


Fig.4.1.1.3. Accuracy and Losses with respect to each epochs

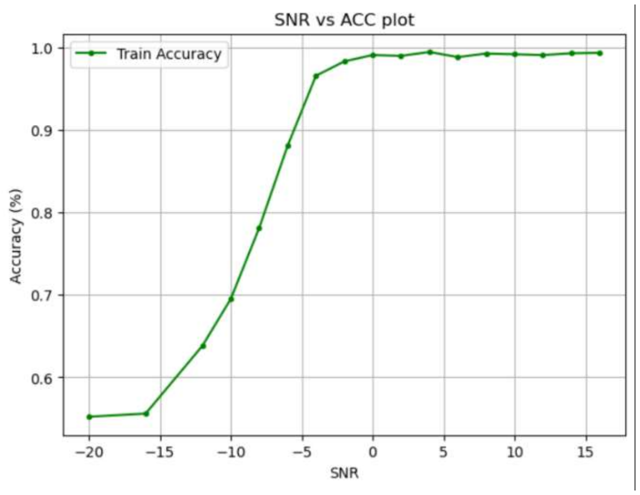
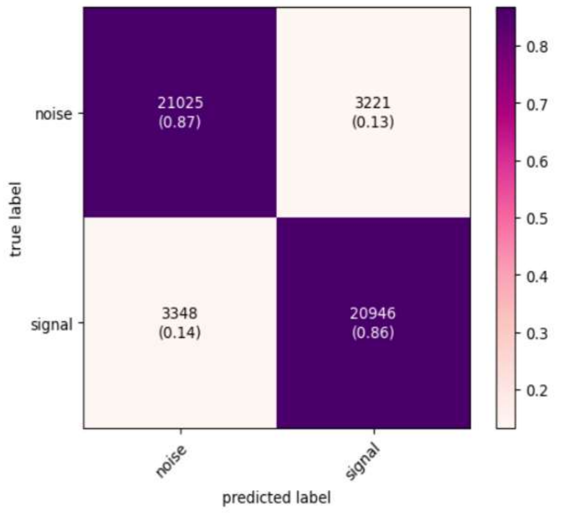
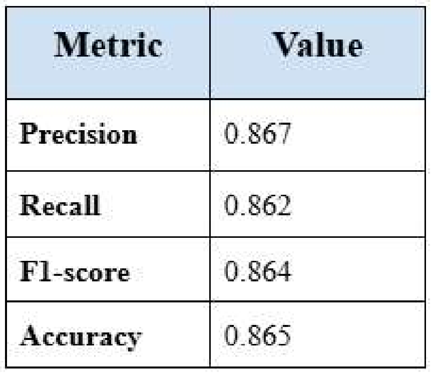


Fig.4.1.1.4. SNR vs ACC plot





Metric	Value
Precision	0.867
Recall	0.862
F1-score	0.864
Accuracy	0.865

Fig.4.1.1.5. Confusion Matrix

Fig.4.1.1.6 Classification Metrics

4.1.2 Confusion Matrix and Classification Report

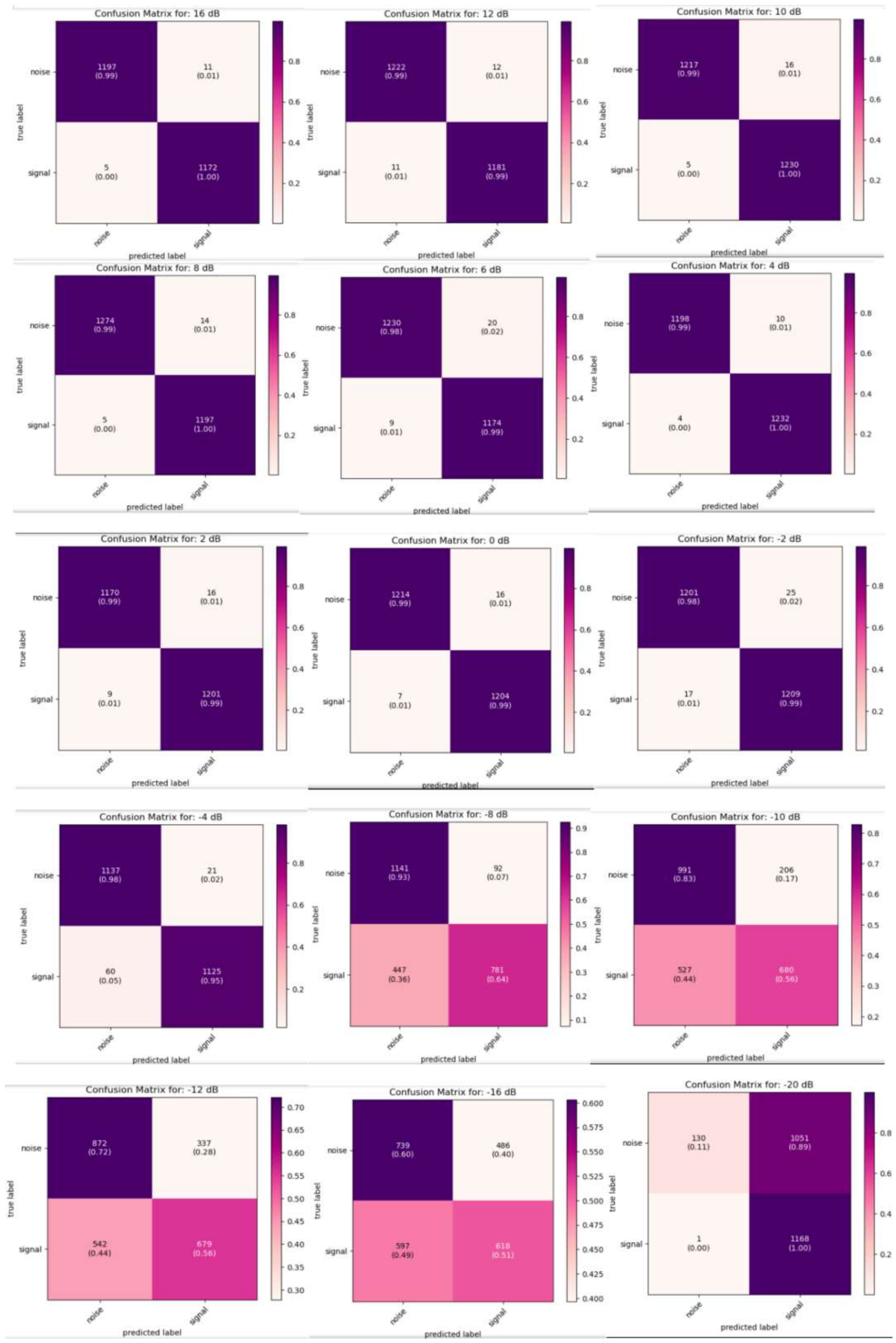


Fig . 4.1.2.1 Confusion Matrix for respective SNR

SNR	Precision	Recall	F1-score
-20	0.5264	0.9991	0.6895
-16	0.558	0.511	0.533
-12	0.67	0.577	0.609
-10	0.77	0.565	0.655
-8	0.894	0.637	0.749
-6	0.97	0.795	0.875
-4	0.982	0.95	0.966
-2	0.98	0.987	0.983
0	0.987	0.994	0.99
2	0.987	0.993	0.99
4	0.992	0.997	0.995
6	0.9833	0.9928	0.9880
8	0.9883	0.9958	0.9920
10	0.9874	0.9959	0.9916
12	0.9900	0.9908	0.9904
14	0.9895	0.9959	0.9927
16	0.9908	0.9958	0.9933

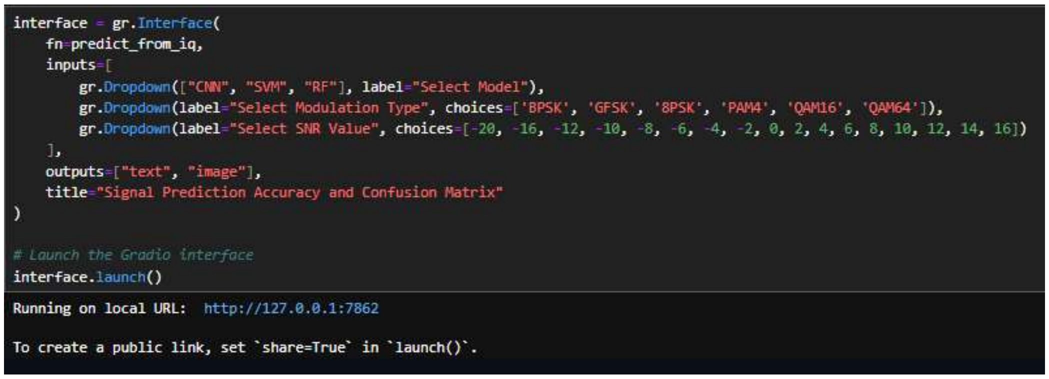
Fig . 4.1.2.2 Classification Report

Metrics Formulae:

- **Accuracy:** ((TP + TN) / (TP + TN + FP + FN))
- **Precision:** (TP / (TP + FP))
- **Recall (Sensitivity):** (TP / (TP + FN))
- **F1 Score:** (2 * (Precision * Recall) / (Precision + Recall))

4.2.2 GUI

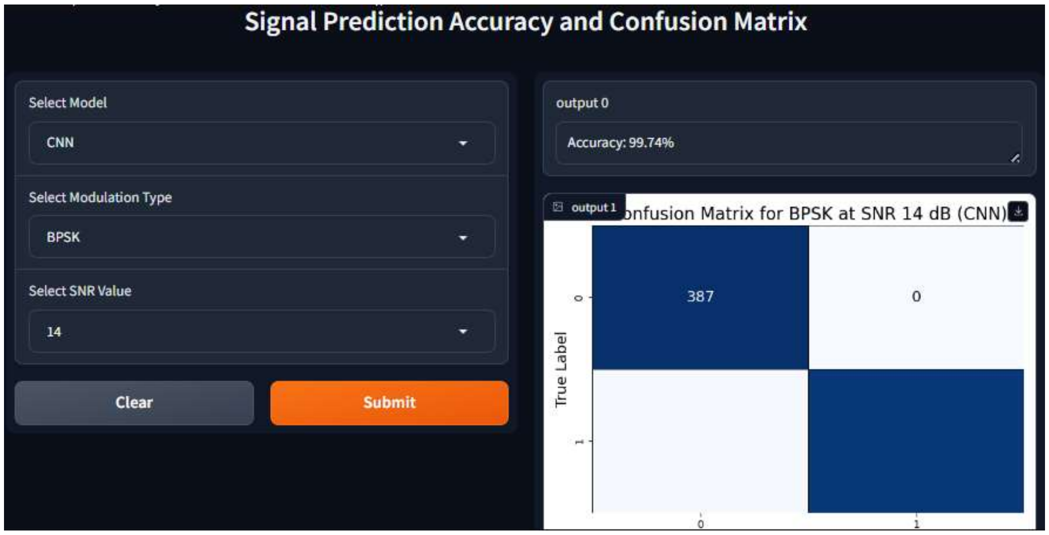
The GUI is built for evaluating signal prediction accuracy and displaying confusion matrices for classification tasks in spectrum sensing. It allows users to interact with machine learning models through dropdown menus, where they can select a specific model (CNN, SVM, RF), modulation type ('BPSK', 'GFSK', '8PSK', 'PAM4', 'QAM16', 'QAM64'), and SNR values ranging from -20 dB to 16 dB. After making selections, the user clicks "Submit" to trigger the model's prediction, which outputs the classification accuracy in a text box and a confusion matrix image that visually compares predicted and true labels. The "Clear" button lets users reset the inputs. The GUI uses Gradio to simplify interaction with signal classification algorithms, providing immediate insights into how well the models perform based on the chosen parameters.



```
interface = gr.Interface(
    fn=predict_from_iq,
    inputs=[
        gr.Dropdown(["CNN", "SVM", "RF"], label="Select Model"),
        gr.Dropdown(label="Select Modulation Type", choices=['BPSK', 'GFSK', '8PSK', 'PAM4', 'QAM16', 'QAM64']),
        gr.Dropdown(label="Select SNR Value", choices=[-20, -16, -12, -10, -8, -6, -4, -2, 0, 2, 4, 6, 8, 10, 12, 14, 16])
    ],
    outputs=["text", "image"],
    title="Signal Prediction Accuracy and Confusion Matrix"
)

# Launch the Gradio interface
interface.launch()
```

```
Running on local URL:  http://127.0.0.1:7862

To create a public link, set `share=True` in `launch()`.
```

Fig . 4.2.2.1 GUI Creation



Signal Prediction Accuracy and Confusion Matrix

Select Model: CNN

Select Modulation Type: BPSK

Select SNR Value: 14

output 0: Accuracy: 99.74%

Fig . 4.2.2.2 Results for CNN

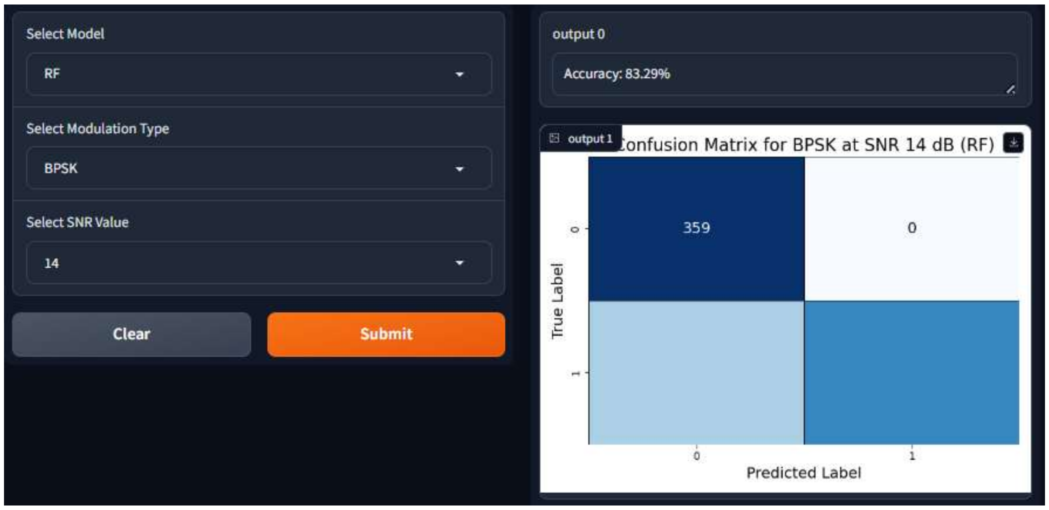

Select Model: RF
Select Modulation Type: BPSK
Select SNR Value: 14
output 0: Accuracy: 83.29%

Fig . 4.2.2.3 Results for RF

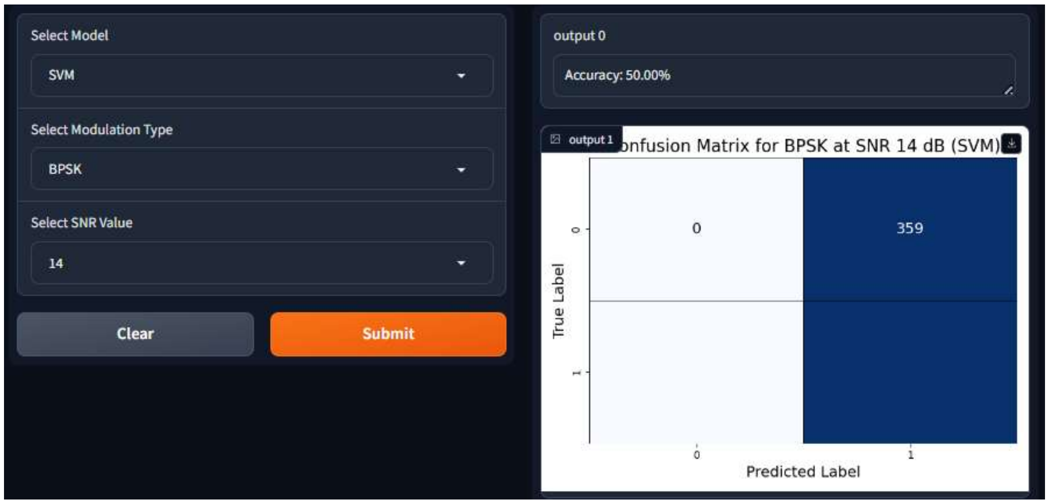

Select Model: SVM
Select Modulation Type: BPSK
Select SNR Value: 14
output 0: Accuracy: 50.00%

Fig . 4.2.2.4 Results for SVM

The results from using CNN, RF, and SVM models to classify BPSK modulation at an SNR of 14 dB reveal distinct performance shows Fig 4.2.2.2 - 4.2.2.4. The CNN model achieved the highest accuracy, 99.74%, indicating its strong ability to distinguish signal features. The RF model, with an accuracy of 83.29%, performed moderately well, showing it can classify the signal effectively, though it may struggle with certain variations or noise. The SVM model performed the worst, with an accuracy of 50%, suggesting it may not handle the complexity of the data as well as the other models. Overall, CNN shows the most promise for accurate signal classification, while RF provides decent performance, and SVM appears less suitable for this task.

4.2. Analysis

The end result of implementation of the three algorithms - Convolution Neural Network, Support Vector Machine, Random Forest Classifier, used for sensing and classification whether the input signal indicates the presence or absence of a primary user in the radio environment generated as a simulation, are tabulated below in the non-increasing order of the metric.

MODEL	ACCURACY
Convolution Neural Network	86.46%
Random Forest	76.21%
Support Vector Machine	49.28%

Fig 4.2.1 Model Accuracy

1. Convolutional Neural Network (CNN):

Achieved the highest accuracy at 86.46%, demonstrating its effectiveness in identifying complex patterns in signal data through automatic feature extraction.The architecture of CNN allows for hierarchical feature learning, which is particularly beneficial for processing multi-dimensional signal inputs.

2. Random Forest (RF):

Achieved an accuracy of 76.21%, performing well but significantly lower than CNN. RF is effective in handling data variability and reducing overfitting.The ensemble approach of RF provides robustness against noise in data, contributing to its ability to generalize better on unseen examples compared to individual classifiers.

3. Support Vector Machine (SVM):

Demonstrated the lowest accuracy of 49.28%, indicating difficulties in managing the complex, nonlinear relationships in the data. While powerful, SVM requires careful tuning to enhance performance. SVM's reliance on a linear hyperplane may limit its effectiveness in high-dimensional spaces without kernel tricks, which necessitates more complex parameter adjustments.

Conclusion:

CNN outperforms both RF and SVM in detecting primary users in simulated radio environments. The significant accuracy gap highlights the importance of algorithm selection based on data characteristics. Future work could focus on hyperparameter tuning for SVM and explore ensemble methods to combine the strengths of these classifiers.