In [7]:
```
import warnings
warnings.filterwarnings('ignore')

# Complex pytorch
import torch
import torch.nn as nn
import torch.nn.functional as F
from torch.utils.data import DataLoader
from torchvision import datasets, transforms
from complexPyTorch.complexLayers import *
from complexPyTorch.complexFunctions import *
from torch_geometric.nn import GCNConv
from torch_geometric.data import Data

# Plot
import matplotlib.pyplot as plt
import seaborn as sns
import time

# Load Data
import numpy as np
import json
import os
import math
import librosa
import pathlib
from scipy.spatial.distance import cdist
from torch.utils.data import Dataset
from sklearn.model_selection import train_test_split
import random
import pandas as pd

# MFCCS
from scipy.io import wavfile
import scipy.fftpack as fft
from scipy.signal import get_window
import librosa
import librosa.display
import IPython.display as ipd
import scipy as spp
```

Create Data

In [ ]:
```
DATASET_PATH = "Data/train"
SAMPLE_RATE = 22050
TRACK_DURATION = 30 # measured in seconds
SAMPLES_PER_TRACK = SAMPLE_RATE * TRACK_DURATION
BATCH_SIZE = 32
NUM_EPOCHS = 50
genre_list = os.listdir(DATASET_PATH)
if '.DS_Store' in genre_list: genre_list.remove('.DS_Store')
genre_mappings = dict(zip(genre_list, range(len(genre_list))))
print(genre_mappings)
```

In [ ]:
```
class TimeDomainFeatures:
    @staticmethod
```

```
    def amplitude_envelope(signal, frame_size, hop_length):
        res = []
        for i in range(0, len(signal), hop_length):
            cur_portion = signal[i:i + frame_size]
            ae_val = max(cur_portion)
            res.append(ae_val)
        return np.array([np.nanmean(res), np.nanvar(res)])

    @staticmethod
    def RMS_energy(signal, frame_size, hop_length):
        res = []
        for i in range(0, len(signal), hop_length):
            cur_portion = signal[i:i + frame_size]
            rmse_val = np.sqrt(1 / len(cur_portion) * sum(i**2 for i in c
            res.append(rmse_val)
        return np.array([np.nanmean(res), np.nanvar(res)])

    @staticmethod
    def crest_factor(signal, frame_size, hop_length):
        res = []
        for i in range(0, len(signal), hop_length):
            cur_portion = signal[i:i + frame_size]
            rmse_val = np.sqrt(1 / len(cur_portion) * sum(i ** 2 for i in
            crest_val = max(np.abs(cur_portion)) / rmse_val
            res.append(crest_val)
        return np.array([np.nanmean(res), np.nanvar(res)])

    @staticmethod
    def ZCR(signal, frame_size, hop_length):
        def num_sign_changes(signal):
            res = 0
            for i in range(0, len(signal) - 1):
                if (signal[i] * signal[i + 1] < 0): res += 1
            return res
        res = []
        for i in range(0, len(signal), hop_length):
            cur_portion = signal[i:i + frame_size]
            zcr_val = num_sign_changes(cur_portion)
            res.append(zcr_val)
        return np.array([np.nanmean(res), np.nanvar(res)])
```

0.1 Real Frequncy Domain

In [ ]:
```
class FreqDomainFeatures:

    @staticmethod
    def normalize_audio(audio):
        audio = audio / np.max(np.abs(audio))
        return audio

    @staticmethod
    def compute_spectrogram(signal, frame_size, hop_length):
        signal = FreqDomainFeatures.normalize_audio(signal)
        spec = librosa.stft(signal, n_fft=frame_size, hop_length=hop_leng
        return np.abs(spec).T

    @staticmethod
    def band_energy_ratio(spec, split_freq = 2048):
```

```
        def find_split_freq_bin(spec, split_freq):
            range_of_freq = SAMPLE_RATE / 2
            change_per_bin = range_of_freq / spec.shape[0]
            split_freq_bin = split_freq / change_per_bin
            return int(np.floor(split_freq_bin))
        split_freq_bin = find_split_freq_bin(spec.T, split_freq)
        res = []
        for sub_arr in spec:
            low_freq_density = sum(i ** 2 for i in sub_arr[:split_freq_bi
            high_freq_density = sum(i ** 2 for i in sub_arr[split_freq_bi
            ber_val = low_freq_density / high_freq_density
            res.append(ber_val)
        return np.array([np.nanmean(res), np.nanvar(res)])

    @staticmethod
    def spectral_centroid(spec):
        def sc(arr):
            res = 0
            for i in range(0, len(arr)):
                res += i*arr[i]
            return res/sum(arr)
        res = []
        for sub_arr in spec:
            sc_val = sc(sub_arr)
            res.append(sc_val)
        return np.array([np.nanmean(res), np.nanvar(res)])

    @staticmethod
    def spectral_bandwidth(spec):
        def sc(arr):
            res = 0
            for i in range(0, len(arr)):
                res += i*arr[i]
            return res/sum(arr)
        def sb(arr):
            res = 0
            sc_val = sc(arr)
            for i in range(0, len(arr)):
                res += (abs(i - sc_val))*arr[i]
            return res/sum(arr)
        res = []
        for sub_arr in spec:
            sb_val = sb(sub_arr)
            res.append(sb_val)
        return np.array([np.nanmean(res), np.nanvar(res)])

    @staticmethod
    def spectral_flatness(spec):
        res = []
        for sub_arr in spec:
            geom_mean = np.exp(np.log(sub_arr).mean())
            ar_mean = np.mean(sub_arr)
            sl_val = geom_mean/ar_mean
            res.append(sl_val)
        return np.array([np.nanmean(res), np.nanvar(res)])
```

In [ ]:

```
class GenreTimeFreqDomain(Dataset):

    def __init__(self, train_path, frame_size=1024, hop_length=512, num_s
        cur_path = pathlib.Path(train_path)
```

```
        self.files = []
        for i in list(cur_path.rglob("*.wav")):
            for j in range(num_segments):
                self.files.append([j, i])
        self.frame_size = frame_size
        self.hop_length = hop_length
        self.training = training
        self.samples_per_segment = int(SAMPLES_PER_TRACK / num_segments)
        self.num_segments = num_segments

    def apply_augmentations(self, signal):
        # Apply augmentations to the audio signal
        if random.random() < 0.5:
            signal = librosa.effects.pitch_shift(signal, sr=SAMPLE_RATE,
        if random.random() < 0.5:
            signal = librosa.effects.time_stretch(signal, rate=random.uni
        return signal

    def __len__(self):
        return len(self.files)

    def adj_shape(self, features):
        if features.shape[0] < 130:
            features = np.pad(features, (0, 130 - features.shape[0]), moc
        else:
            features = features[:130]
        return features

    def get_time_domain(self, cur_signal):
        ae = TimeDomainFeatures.amplitude_envelope(cur_signal, self.frame
        rmse = TimeDomainFeatures.RMS_energy(cur_signal, self.frame_size,
        cf = TimeDomainFeatures.crest_factor(cur_signal, self.frame_size,
        zcr = TimeDomainFeatures.ZCR(cur_signal, self.frame_size, self.hc
        return np.concatenate([ae, rmse, cf, zcr])

    def get_freq_domain(self, cur_signal):
        spec = FreqDomainFeatures.compute_spectrogram(cur_signal, self.fr
        ber = FreqDomainFeatures.band_energy_ratio(spec)
        sc = FreqDomainFeatures.spectral_centroid(spec)
        sb = FreqDomainFeatures.spectral_bandwidth(spec)
        sf = FreqDomainFeatures.spectral_flatness(spec)
        return np.concatenate([ber, sc, sb, sf])


    def __getitem__(self, idx):
        cur_file = self.files[idx]
        d = cur_file[0]
        file_path = cur_file[1]
        target = genre_mappings[str(file_path).split("/")[2]]
        signal, sample_rate = librosa.load(file_path, sr=SAMPLE_RATE)
        start = self.samples_per_segment * d
        finish = start + self.samples_per_segment
        cur_signal = signal[start:finish]
        # if self.training: cur_signal = self.apply_augmentations(cur_sig
        td_features = self.get_time_domain(cur_signal)
        fd_features = self.get_freq_domain(cur_signal)
        return torch.tensor(np.array([td_features, fd_features]).flatten(
```

In [ ]:

```
train_dataset = GenreTimeFreqDomain("Data/train/", training = True)
test_dataset = GenreTimeFreqDomain("Data/test/", training = False)
```

```
column_names = []
for j in ["ae", "rmse", "cf", "zcr", "ber", "sc", "sb", "sf"]:
    for i in ["mean", "var"]:
        column_names.extend([f"{j}_{i}"])
# Add label and set columns
column_names.extend(['label'])
print(column_names)


def create_dataframe(dataset):
    features_list = []
    labels_list = []
    for i in range(len(dataset)):
        features, label = dataset[i]
        features_list.append(features)
        labels_list.append(label)
        if i%10 == 0: print(i)
    df = pd.DataFrame(features_list) # Flatten the features
    df['label'] = labels_list
    return df

train_df = create_dataframe(train_dataset)
train_df.columns = column_names
print("-"*75)
test_df = create_dataframe(test_dataset)
test_df.columns = column_names

tr_df = train_df.applymap(lambda x: x.numpy() if hasattr(x, 'numpy') else
te_df = test_df.applymap(lambda x: x.numpy() if hasattr(x, 'numpy') else
tr_df.to_csv("train_tff.csv", index = False)
te_df.to_csv("test_tff.csv", index = False)
```

0.2 Complex Frequncy Domain

In [ ]:

```
class FreqDomainFeatures:

    @staticmethod
    def normalize_audio(audio):
        audio = audio / np.max(np.abs(audio))
        return audio

    @staticmethod
    def compute_spectrogram(signal, frame_size, hop_length):
        signal = FreqDomainFeatures.normalize_audio(signal)
        spec = librosa.stft(signal, n_fft=frame_size, hop_length=hop_leng
        return (spec).T

    @staticmethod
    def band_energy_ratio(spec, split_freq = 2048):
        def find_split_freq_bin(spec, split_freq):
            range_of_freq = SAMPLE_RATE / 2
            change_per_bin = range_of_freq / spec.shape[0]
            split_freq_bin = split_freq / change_per_bin
            return int(np.floor(split_freq_bin))
        split_freq_bin = find_split_freq_bin(spec.T, split_freq)
        res = []
        for sub_arr in spec:
```

```
            low_freq_density = sum(i ** 2 for i in sub_arr[:split_freq_bi
            high_freq_density = sum(i ** 2 for i in sub_arr[split_freq_bi
            ber_val = low_freq_density / high_freq_density
            res.append(ber_val)
        return np.array([np.nanmean(res), np.nanvar(res)])

    @staticmethod
    def spectral_centroid(spec):
        def sc(arr):
            res = 0
            for i in range(0, len(arr)):
                res += i*arr[i]
            return res/sum(arr)
        res = []
        for sub_arr in spec:
            sc_val = sc(sub_arr)
            res.append(sc_val)
        return np.array([np.nanmean(res), np.nanvar(res)])

    @staticmethod
    def spectral_bandwidth(spec):
        def sc(arr):
            res = 0
            for i in range(0, len(arr)):
                res += i*arr[i]
            return res/sum(arr)
        def sb(arr):
            res = 0
            sc_val = sc(arr)
            for i in range(0, len(arr)):
                res += (np.abs(i - sc_val))*arr[i]
            return res/sum(arr)
        res = []
        for sub_arr in spec:
            sb_val = sb(sub_arr)
            res.append(sb_val)
        return np.array([np.nanmean(res), np.nanvar(res)])

    @staticmethod
    def spectral_flatness(spec):
        res = []
        for sub_arr in spec:
            geom_mean = np.exp(np.log(sub_arr).mean())
            ar_mean = np.mean(sub_arr)
            sl_val = geom_mean/ar_mean
            res.append(sl_val)
        return np.array([np.nanmean(res), np.nanvar(res)])
```

In [ ]:

```
class GenreTimeFreqDomain(Dataset):

    def __init__(self, train_path, frame_size=1024, hop_length=512, num_s
        cur_path = pathlib.Path(train_path)
        self.files = []
        for i in list(cur_path.rglob("*.wav")):
            for j in range(num_segments):
                self.files.append([j, i])
        self.frame_size = frame_size
        self.hop_length = hop_length
        self.training = training
        self.samples_per_segment = int(SAMPLES_PER_TRACK / num_segments)
```

```
        self.num_segments = num_segments

    def apply_augmentations(self, signal):
        # Apply augmentations to the audio signal
        if random.random() < 0.5:
            signal = librosa.effects.pitch_shift(signal, sr=SAMPLE_RATE,
        if random.random() < 0.5:
            signal = librosa.effects.time_stretch(signal, rate=random.uni
        return signal

    def __len__(self):
        return len(self.files)

    def adj_shape(self, features):
        if features.shape[0] < 130:
            features = np.pad(features, (0, 130 - features.shape[0]), moc
        else:
            features = features[:130]
        return features

    def get_time_domain(self, cur_signal):
        ae = TimeDomainFeatures.amplitude_envelope(cur_signal, self.frame
        rmse = TimeDomainFeatures.RMS_energy(cur_signal, self.frame_size,
        cf = TimeDomainFeatures.crest_factor(cur_signal, self.frame_size,
        zcr = TimeDomainFeatures.ZCR(cur_signal, self.frame_size, self.hc
        return np.concatenate([ae, rmse, cf, zcr])

    def get_freq_domain(self, cur_signal):
        spec = FreqDomainFeatures.compute_spectrogram(cur_signal, self.fr
        ber = FreqDomainFeatures.band_energy_ratio(spec)
        sc = FreqDomainFeatures.spectral_centroid(spec)
        sb = FreqDomainFeatures.spectral_bandwidth(spec)
        sf = FreqDomainFeatures.spectral_flatness(spec)
        return np.concatenate([ber, sc, sb, sf])


    def __getitem__(self, idx):
        cur_file = self.files[idx]
        d = cur_file[0]
        file_path = cur_file[1]
        target = genre_mappings[str(file_path).split("/")[2]]
        signal, sample_rate = librosa.load(file_path, sr=SAMPLE_RATE)
        start = self.samples_per_segment * d
        finish = start + self.samples_per_segment
        cur_signal = signal[start:finish]
        # if self.training: cur_signal = self.apply_augmentations(cur_sig
        td_features = self.get_time_domain(cur_signal)
        fd_features = self.get_freq_domain(cur_signal)
        return torch.tensor(np.array([td_features, fd_features]).flatten(
```

In [ ]:

```
train_dataset = GenreTimeFreqDomain("Data/train/", training = True)
test_dataset = GenreTimeFreqDomain("Data/test/", training = False)

column_names = []
for j in ["ae", "rmse", "cf", "zcr", "ber", "sc", "sb", "sf"]:
    for i in ["mean", "var"]:
        column_names.extend([f"{j}_{i}"])
# Add label and set columns
column_names.extend(['label'])
print(column_names)
```

```
def create_dataframe(dataset):
    features_list = []
    labels_list = []
    for i in range(len(dataset)):
        features, label = dataset[i]
        features_list.append(features)
        labels_list.append(label)
        if i%10 == 0: print(i)
    df = pd.DataFrame(features_list) # Flatten the features
    df['label'] = labels_list
    return df

train_df = create_dataframe(train_dataset)
train_df.columns = column_names
print("-"*75)
test_df = create_dataframe(test_dataset)
test_df.columns = column_names

tr_df = train_df.applymap(lambda x: x.numpy() if hasattr(x, 'numpy') else
te_df = test_df.applymap(lambda x: x.numpy() if hasattr(x, 'numpy') else
tr_df.to_csv("train_tff_comp.csv", index = False)
te_df.to_csv("test_tff_comp.csv", index = False)
```

1. Simple Decsion Tree with Real Valued Frequency domain Features

In [8]:
```
tr_df = pd.read_csv("train_tff.csv")
te_df = pd.read_csv("test_tff.csv")
```

In [9]:
```
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression, SGDClassifier, Ridge
from sklearn.neighbors import KNeighborsClassifier, RadiusNeighborsClassi
from sklearn.naive_bayes import GaussianNB, MultinomialNB, ComplementNB,
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier, AdaBoostClassifier,
from sklearn.svm import SVC, NuSVC, LinearSVC
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis, Qua
from sklearn.ensemble import VotingClassifier
from sklearn.metrics import accuracy_score

# Separate features and labels
X_train = tr_df.drop('label', axis=1)
y_train = tr_df['label']
X_test = te_df.drop('label', axis=1)
y_test = te_df['label']

# List of models to try
models = [
    ('Logistic Regression', LogisticRegression()),
    ('SGD Classifier', SGDClassifier()),
    ('Ridge Classifier', RidgeClassifier()),
    ('Passive Aggressive Classifier', PassiveAggressiveClassifier()),
    ('K-Nearest Neighbors', KNeighborsClassifier()),
    ('Gaussian Naive Bayes', GaussianNB()),
    ('Multinomial Naive Bayes', MultinomialNB()),
```

```
    ('Complement Naive Bayes', ComplementNB()),
    ('Bernoulli Naive Bayes', BernoulliNB()),
    ('Decision Tree', DecisionTreeClassifier()),
    ('Random Forest', RandomForestClassifier()),
    ('AdaBoost', AdaBoostClassifier()),
    ('Gradient Boosting', GradientBoostingClassifier()),
    ('Support Vector Machine', SVC()),
    ('Nu-Support Vector Machine', NuSVC()),
    ('Linear Support Vector Machine', LinearSVC()),
    ('Linear Discriminant Analysis', LinearDiscriminantAnalysis()),
    ('Quadratic Discriminant Analysis', QuadraticDiscriminantAnalysis()),
    ('Voting Classifier', VotingClassifier(estimators=[
        ('lr', LogisticRegression()),
        ('rf', RandomForestClassifier()),
        ('svc', SVC())
    ]))
    # Add more models as needed
]

# Loop through models
for model_name, model in models:
    # Train the model
    model.fit(X_train, y_train)

    # Predict on test data
    y_pred = model.predict(X_test)

    # Calculate accuracy
    accuracy = accuracy_score(y_test, y_pred)

    # Print results
    print(f'Model: {model_name}')
    print(f'Accuracy: {accuracy}\n')
```

```
Model: Logistic Regression
Accuracy: 0.11625

Model: SGD Classifier
Accuracy: 0.05875

Model: Ridge Classifier
Accuracy: 0.364375

Model: Passive Aggressive Classifier
Accuracy: 0.21625

Model: K-Nearest Neighbors
Accuracy: 0.2625

Model: Gaussian Naive Bayes
Accuracy: 0.113125

Model: Multinomial Naive Bayes
Accuracy: 0.145

Model: Complement Naive Bayes
Accuracy: 0.110625

Model: Bernoulli Naive Bayes
Accuracy: 0.1

Model: Decision Tree
Accuracy: 0.4125

Model: Random Forest
Accuracy: 0.5425

Model: AdaBoost
Accuracy: 0.420625

Model: Gradient Boosting
Accuracy: 0.539375

Model: Support Vector Machine
Accuracy: 0.099375

Model: Nu-Support Vector Machine
Accuracy: 0.110625

Model: Linear Support Vector Machine
Accuracy: 0.11

Model: Linear Discriminant Analysis
Accuracy: 0.421875

Model: Quadratic Discriminant Analysis
Accuracy: 0.34375

Model: Voting Classifier
Accuracy: 0.18375
```

In [33]:

```
class Node:
    def __init__(self, feature=None, threshold=None, left=None, right=Non
```

```
        self.feature = feature
        self.threshold = threshold
        self.left = left
        self.right = right
        self.value = value

    def is_leaf(self):
        return self.value is not None

class DecisionTree:
    def __init__(self, max_depth=100, min_samples_split=2):
        self.max_depth = max_depth
        self.min_samples_split = min_samples_split
        self.root = None

    def _is_finished(self, depth):
        if (depth >= self.max_depth
            or self.n_class_labels == 1
            or self.n_samples < self.min_samples_split):
            return True
        return False

    def _entropy(self, y):
        proportions = np.bincount(y) / len(y)
        entropy = -np.sum([p * np.log2(p) for p in proportions if p > 0])
        return entropy

    def _create_split(self, X, thresh):
        left_idx = np.argwhere(X <= thresh).flatten()
        right_idx = np.argwhere(X > thresh).flatten()
        return left_idx, right_idx

    def _information_gain(self, X, y, thresh):
        parent_loss = self._entropy(y)
        left_idx, right_idx = self._create_split(X, thresh)
        n, n_left, n_right = len(y), len(left_idx), len(right_idx)

        if n_left == 0 or n_right == 0:
            return 0

        child_loss = (n_left / n) * self._entropy(y[left_idx]) + (n_right
        return parent_loss - child_loss

    def _best_split(self, X, y, features):
        split = {'score':- 1, 'feat': None, 'thresh': None}

        for feat in features:
            X_feat = X[:, feat]
            thresholds = np.unique(X_feat)
            for thresh in thresholds:
                score = self._information_gain(X_feat, y, thresh)

                if score > split['score']:
                    split['score'] = score
                    split['feat'] = feat
                    split['thresh'] = thresh

        return split['feat'], split['thresh']

    def _build_tree(self, X, y, depth=0):
```

```
        self.n_samples, self.n_features = X.shape
        self.n_class_labels = len(np.unique(y))

        # stopping criteria
        if self._is_finished(depth):
            most_common_Label = np.argmax(np.bincount(y))
            return Node(value=most_common_Label)

        # get best split
        rnd_feats = np.random.choice(self.n_features, self.n_features, re
        best_feat, best_thresh = self._best_split(X, y, rnd_feats)

        # grow children recursively
        left_idx, right_idx = self._create_split(X[:, best_feat], best_th
        left_child = self._build_tree(X[left_idx, :], y[left_idx], depth
        right_child = self._build_tree(X[right_idx, :], y[right_idx], dep
        return Node(best_feat, best_thresh, left_child, right_child)

    def _traverse_tree(self, x, node):
        if node.is_leaf():
            return node.value

        if x[node.feature] <= node.threshold:
            return self._traverse_tree(x, node.left)
        return self._traverse_tree(x, node.right)

    def fit(self, X, y):
        self.root = self._build_tree(X, y)

    def predict(self, X):
        predictions = [self._traverse_tree(x, self.root) for x in X]
        return np.array(predictions)
```

In [35]:
```
np.random.seed(42)
model = DecisionTree(max_depth=10)
model.fit(X_train.to_numpy(), y_train.to_numpy())
y_pred = model.predict(X_test.to_numpy())
accuracy = accuracy_score(y_test.to_numpy(), y_pred)
print(f'Accuracy: {accuracy}\n')
```

```
Accuracy: 0.451875
```

In [36]:
```
np.random.seed(42)
model = DecisionTree(max_depth=10)
model.fit(X_train.to_numpy(), y_train.to_numpy())
y_pred = model.predict(X_test.to_numpy())
accuracy = accuracy_score(y_test.to_numpy(), y_pred)
print(f'Accuracy: {accuracy}\n')
```

```
Accuracy: 0.451875
```

2. Simple Decision Tree with Complex Valued Frequency Domain Features

In [131…
```
tr_df = pd.read_csv("train_tff_comp.csv")
te_df = pd.read_csv("test_tff_comp.csv")
```

```
import pandas as pd

def df_csv_complex(df):
    result_df = df.copy()  # Make a copy to avoid modifying the original
    result_df.iloc[:, :-1] = df.iloc[:, :-1].apply(lambda col: col.apply(
    return result_df

tr_df = df_csv_complex(tr_df)
te_df = df_csv_complex(te_df)
```

2.1 Compare only real

In [77]:
```
class Node:
    def __init__(self, feature=None, threshold=None, left=None, right=Non
        self.feature = feature
        self.threshold = threshold
        self.left = left
        self.right = right
        self.value = value

    def is_leaf(self):
        return self.value is not None

class DecisionTree:
    def __init__(self, max_depth=100, min_samples_split=2):
        self.max_depth = max_depth
        self.min_samples_split = min_samples_split
        self.root = None

    def _is_finished(self, depth):
        if (depth >= self.max_depth
            or self.n_class_labels == 1
            or self.n_samples < self.min_samples_split):
            return True
        return False

    def _entropy(self, y):
        proportions = np.bincount(y) / len(y)
        entropy = -np.sum([p * np.log2(p) for p in proportions if p > 0])
        return entropy

    def _create_split(self, X, thresh):
        left_idx = np.argwhere(X <= thresh).flatten()
        right_idx = np.argwhere(X > thresh).flatten()
        return left_idx, right_idx

    def _information_gain(self, X, y, thresh):
        parent_loss = self._entropy(y)
        left_idx, right_idx = self._create_split(X, thresh)
        n, n_left, n_right = len(y), len(left_idx), len(right_idx)

        if n_left == 0 or n_right == 0:
            return 0

        child_loss = (n_left / n) * self._entropy(y[left_idx]) + (n_right
        return parent_loss - child_loss
```

```
    def _best_split(self, X, y, features):
        split = {'score':- 1, 'feat': None, 'thresh': None}

        for feat in features:
            X_feat = X[:, feat]
            thresholds = np.unique(X_feat)
            for thresh in thresholds:
                score = self._information_gain(X_feat, y, thresh)

                if score > split['score']:
                    split['score'] = score
                    split['feat'] = feat
                    split['thresh'] = thresh

        return split['feat'], split['thresh']

    def _build_tree(self, X, y, depth=0):
        self.n_samples, self.n_features = X.shape
        self.n_class_labels = len(np.unique(y))

        # stopping criteria
        if self._is_finished(depth):
            most_common_Label = np.argmax(np.bincount(y))
            return Node(value=most_common_Label)

        # get best split
        rnd_feats = np.random.choice(self.n_features, self.n_features, re
        best_feat, best_thresh = self._best_split(X, y, rnd_feats)

        # grow children recursively
        left_idx, right_idx = self._create_split(X[:, best_feat], best_th
        left_child = self._build_tree(X[left_idx, :], y[left_idx], depth
        right_child = self._build_tree(X[right_idx, :], y[right_idx], dep
        return Node(best_feat, best_thresh, left_child, right_child)

    def _traverse_tree(self, x, node):
        if node.is_leaf():
            return node.value

        if x[node.feature] <= node.threshold:
            return self._traverse_tree(x, node.left)
        return self._traverse_tree(x, node.right)

    def fit(self, X, y):
        self.root = self._build_tree(X, y)

    def predict(self, X):
        predictions = [self._traverse_tree(x, self.root) for x in X]
        return np.array(predictions)
```

In [67]:

```
# Separate features and labels
X_train = tr_df.drop('label', axis=1)
y_train = tr_df['label']
X_test = te_df.drop('label', axis=1)
y_test = te_df['label']
```

In [68]:

```
np.random.seed(42)
model = DecisionTree(max_depth=10)
model.fit(X_train.to_numpy(), y_train.to_numpy())
```

```
y_pred = model.predict(X_test.to_numpy())
accuracy = accuracy_score(y_test.to_numpy(), y_pred)
print(f'Accuracy: {accuracy}\n')
```

```
Accuracy: 0.4575
```

2.2 Compare only magnitude

In [93]:
```
class Node:
    def __init__(self, feature=None, threshold=None, left=None, right=Non
        self.feature = feature
        self.threshold = threshold
        self.left = left
        self.right = right
        self.value = value

    def is_leaf(self):
        return self.value is not None

class DecisionTree:
    def __init__(self, max_depth=100, min_samples_split=2):
        self.max_depth = max_depth
        self.min_samples_split = min_samples_split
        self.root = None

    def _is_finished(self, depth):
        if (depth >= self.max_depth
            or self.n_class_labels == 1
            or self.n_samples < self.min_samples_split):
            return True
        return False

    def _entropy(self, y):
        proportions = np.bincount(y) / len(y)
        entropy = -np.sum([p * np.log2(p) for p in proportions if p > 0])
        return entropy

    def _create_split(self, X, thresh):
        left_idx = np.argwhere(np.abs(X) <= np.abs(thresh)).flatten()
        right_idx = np.argwhere(np.abs(X) > np.abs(thresh)).flatten()
        return left_idx, right_idx

    def _information_gain(self, X, y, thresh):
        parent_loss = self._entropy(y)
        left_idx, right_idx = self._create_split(X, thresh)
        n, n_left, n_right = len(y), len(left_idx), len(right_idx)

        if n_left == 0 or n_right == 0:
            return 0

        child_loss = (n_left / n) * self._entropy(y[left_idx]) + (n_right
        return parent_loss - child_loss

    def _best_split(self, X, y, features):
        split = {'score':- 1, 'feat': None, 'thresh': None}

        for feat in features:
            X_feat = X[:, feat]
```

```python
            thresholds = np.unique(X_feat)
            for thresh in thresholds:
                score = self._information_gain(X_feat, y, thresh)

                if np.abs(score) > np.abs(split['score']) if split['score
                    split['score'] = score
                    split['feat'] = feat
                    split['thresh'] = thresh

        return split['feat'], split['thresh']

    def _build_tree(self, X, y, depth=0):
        self.n_samples, self.n_features = X.shape
        self.n_class_labels = len(np.unique(y))

        # stopping criteria
        if self._is_finished(depth):
            most_common_Label = np.argmax(np.bincount(y))
            return Node(value=most_common_Label)

        # get best split
        rnd_feats = np.random.choice(self.n_features, self.n_features, re
        best_feat, best_thresh = self._best_split(X, y, rnd_feats)

        # grow children recursively
        left_idx, right_idx = self._create_split(X[:, best_feat], best_th
        left_child = self._build_tree(X[left_idx, :], y[left_idx], depth
        right_child = self._build_tree(X[right_idx, :], y[right_idx], dep
        return Node(best_feat, best_thresh, left_child, right_child)

    def _traverse_tree(self, x, node):
        if node.is_leaf():
            return node.value

        if x[node.feature] <= node.threshold:
            return self._traverse_tree(x, node.left)
        return self._traverse_tree(x, node.right)

    def fit(self, X, y):
        self.root = self._build_tree(X, y)

    def predict(self, X):
        predictions = [self._traverse_tree(x, self.root) for x in X]
        return np.array(predictions)
```

In [94]:

```python
np.random.seed(42)
model = DecisionTree(max_depth=10)
model.fit(X_train.to_numpy(), y_train.to_numpy())
y_pred = model.predict(X_test.to_numpy())
accuracy = accuracy_score(y_test.to_numpy(), y_pred)
print(f'Accuracy: {accuracy}\n')
```

```
Accuracy: 0.3975
```

In [ ]:

In [ ]: