Practical No :5

Aim: Generative Models

a. Implement and demonstrate the working of a Naïve Bayesian classifier using a sample data set. Build the model to classify a test sample.

Description:

The **Naïve Bayes classifier** is a simple probabilistic machine learning model based on **Bayes' Theorem**. It is called *naïve* because it assumes that all features are **independent** of each other given the class label, which rarely holds true in real datasets but works surprisingly well in practice.

$$P(C|X) = \frac{P(X|C)\cdot P(C)}{P(X)}$$

Where:

- P(C|X)P(C|X)P(C|X) → Posterior probability (probability of class given features)
- P(C)P(C)P(C) → Prior probability of class
- P(X|C)P(X|C)P(X|C) → Likelihood (probability of features given class)
- P(X)P(X)P(X) → Evidence (normalizing constant)

2. Working of Naïve Bayes

1. **Training Phase**
   - Calculate prior probabilities for each class.
   - For each feature, calculate conditional probabilities P(xi|C)P(x_i | C)P(xi|C).
2. **Prediction Phase**
   - For a new test sample X=(x1,x2,…,xn)X = (x_1, x_2, …, x_n)X=(x1,x2,…,xn), compute:

$$P(C|X) \propto P(C) \cdot \prod_{i=1}^{n} P(x_i|C)$$

Code:

```
import pandas as pd
import numpy as np
from collections import defaultdict
import math

class NaiveBayesClassifier:

  def __init__(self):
```

```
        self.class_probs = {}
        self.feature_probs = defaultdict(lambda: defaultdict(lambda: defaultdict(float)))
        self.features = []
    def fit(self, df: pd.DataFrame, target_column: str):
        self.features = [col for col in df.columns if col != target_column]
        total_samples = len(df)
        # Calculate prior probabilities
        class_counts = df[target_column].value_counts()
        self.class_probs = (class_counts / total_samples).to_dict()
        # Calculate conditional probabilities with Laplace smoothing
        for feature in self.features:
            for target_class in class_counts.index:
                subset = df[df[target_column] == target_class]
                value_counts = subset[feature].value_counts()
                unique_values = df[feature].unique()
                for value in unique_values:
                    count = value_counts.get(value, 0)
                    # Laplace smoothing
                    prob = (count + 1) / (len(subset) + len(unique_values))
                    self.feature_probs[feature][value][target_class] = prob
    def predict(self, input_sample: dict):
        results = {}
        for target_class in self.class_probs:
            log_prob = math.log(self.class_probs[target_class])
            for feature, value in input_sample.items():
                if value in self.feature_probs[feature]:
```

```
                log_prob += math.log(self.feature_probs[feature][value][target_class])
            else:
                # Handle unknown values with small smoothing
                log_prob += math.log(1e-6)
        results[target_class] = log_prob
    # Normalize to probabilities
    max_log = max(results.values())
    probs_exp = {cls: math.exp(val - max_log) for cls, val in results.items()}
    total = sum(probs_exp.values())
    probs = {cls: val / total for cls, val in probs_exp.items()}
    return max(probs, key=probs.get), probs
  def print_model(self):
    print("Prior probabilities:")
    for cls, prob in self.class_probs.items():
      print(f"P({cls}) = {prob:.4f}")
    print("\nConditional probabilities:")
    for feature in self.features:
      print(f"\nFeature: {feature}")
      for value in self.feature_probs[feature]:
        for cls in self.class_probs:
          prob = self.feature_probs[feature][value][cls]
          print(f"P({feature}={value} | {cls}) = {prob:.4f}")

# --- MAIN CODE ---
if __name__ == "__main__":
  # Load dataset
```

```
df = pd.read_csv("sample_data.csv")
# Initialize and train model
model = NaiveBayesClassifier()
model.fit(df, target_column="PlayTennis")
# Optional: print learned model
model.print_model()
# Define test sample
test_sample = {
    "Outlook": "Sunny",
    "Temperature": "Cool",
    "Humidity": "High",
    "Wind": "Strong"
}
# Predict
prediction, probabilities = model.predict(test_sample)
print(f"\nTest Sample: {test_sample}")
print(f"Predicted Class: {prediction}")
print("Class Probabilities:")
for cls, prob in probabilities.items():
    print(f"{cls}: {prob:.4f}")
```

Output:

```
Prior probabilities:
P(Yes) = 0.6154
P(No) = 0.3846

Conditional probabilities:

Feature: Outlook
P(Outlook=Sunny | Yes) = 0.1667
P(Outlook=Sunny | No) = 0.4444
P(Outlook=Overcast | Yes) = 0.4167
P(Outlook=Overcast | No) = 0.1111
P(Outlook=Rain | Yes) = 0.3333
P(Outlook=Rain | No) = 0.2222
P(Outlook=Rain  | Yes) = 0.0833
P(Outlook=Rain  | No) = 0.2222

Feature: Temperature
P(Temperature=Hot | Yes) = 0.2727
P(Temperature=Hot | No) = 0.3750
P(Temperature=Mild | Yes) = 0.4545
P(Temperature=Mild | No) = 0.3750
P(Temperature=Cool | Yes) = 0.2727
P(Temperature=Cool | No) = 0.2500

Feature: Humidity
P(Humidity=High | Yes) = 0.4000
P(Humidity=High | No) = 0.7143
P(Humidity=Normal | Yes) = 0.6000
P(Humidity=Normal | No) = 0.2857

Feature: Wind
P(Wind=Weak | Yes) = 0.5000
P(Wind=Weak | No) = 0.4286
P(Wind=Strong | Yes) = 0.5000
P(Wind=Strong | No) = 0.5714

Test Sample: {'Outlook': 'Sunny', 'Temperature': 'Cool', 'Humidity': 'High', 'Wind': 'Strong'}
Predicted Class: No
Class Probabilities:
Yes: 0.2428
No: 0.7572
```

Learnings:

Naïve Bayes is a probability-based classifier that predicts outcomes using past data.

It assumes all features are independent of each other within a class.

During training, it learns how often each class and feature value occurs.

For a new test sample, it calculates which class is most likely.

Aim: b. Implement Hidden Markov Models using hmmlearn

Description:

Hidden Markov Model (HMM) – Description

A **Hidden Markov Model (HMM)** is a statistical model that assumes a system can be described by:

Hidden states (unobserved, e.g., weather conditions, user intent).

Observations (visible outcomes influenced by states, e.g., activities, words).

Probabilities for state transitions and for generating observations.

The goal is to use observed data to **infer the most likely sequence of hidden states** or to **predict future observations**.

Using hmmlearn

The **hmmlearn library** in Python provides ready-to-use implementations of HMMs.

Steps followed in implementation:

Import library (from hmmlearn import hmm).

Prepare dataset – observations are usually integers or encoded values.

Define HMM model – for example, GaussianHMM for continuous data or MultinomialHMM for categorical data.

Train the model using fit() with observation sequences.

Predict hidden states using predict() and evaluate results.

Code:

```
import numpy as np

import pandas as pd

from sklearn.preprocessing import LabelEncoder

import matplotlib.pyplot as plt

import traceback


"""

Rewritten HMM example for the PlayTennis CSV.

This version is more robust and fixes common errors:

 - checks required columns
```

```
 - uses tuple-to-index mapping for categorical observations
 - uses a scaled forward algorithm to avoid underflow (returns log-likelihood)
 - uses Viterbi implemented in log-space for numerical stability
 - prints helpful debug information

To run: make sure the CSV is at /mnt/data/sample_data.csv (the uploaded file path).
"""

try:
    # --- Load ---
    df = pd.read_csv("sample_data.csv")
    print("Loaded sample_data.csv with shape:", df.shape)

    # expected columns in the classic PlayTennis dataset
    required_cols = ["Outlook", "Temperature", "Humidity", "Wind", "PlayTennis"]
    missing = [c for c in required_cols if c not in df.columns]
    if missing:
        raise ValueError(f"Missing required columns in CSV: {missing}")

    # --- Encode categorical columns ---
    encoders = {}
    for col in required_cols:
        le = LabelEncoder()
        # convert to string first to avoid issues with unexpected dtypes
        df[col] = le.fit_transform(df[col].astype(str))
        encoders[col] = le

    # Features and labels
    X = df[["Outlook", "Temperature", "Humidity", "Wind"]].values.astype(int)
```

```
y = df["PlayTennis"].values.astype(int)

# Build observation tuples and map them to indices
obs_tuples = [tuple(row) for row in X]
unique_obs = sorted(list(set(obs_tuples)))
obs_map = {tup: i for i, tup in enumerate(unique_obs)}
obs_seq_idx = np.array([obs_map[t] for t in obs_tuples], dtype=int)

n_obs = len(unique_obs)
n_states = 2  # you can change this if you want more hidden states
T = len(obs_seq_idx)

print(f"Number of unique observations: {n_obs}")
print(f"Sequence length: {T}, Hidden states: {n_states}")

# --- Initialize model params ---
np.random.seed(42)
start_prob = np.full(n_states, 1.0 / n_states)

trans_prob = np.random.rand(n_states, n_states)
trans_prob /= trans_prob.sum(axis=1, keepdims=True)

emit_prob = np.random.rand(n_states, n_obs)
emit_prob /= emit_prob.sum(axis=1, keepdims=True)

# --- Forward algorithm with scaling (returns log-likelihood) ---
def forward_scaled(obs_idx, start_p, trans_p, emit_p):
    T = len(obs_idx)
    N = len(start_p)
```

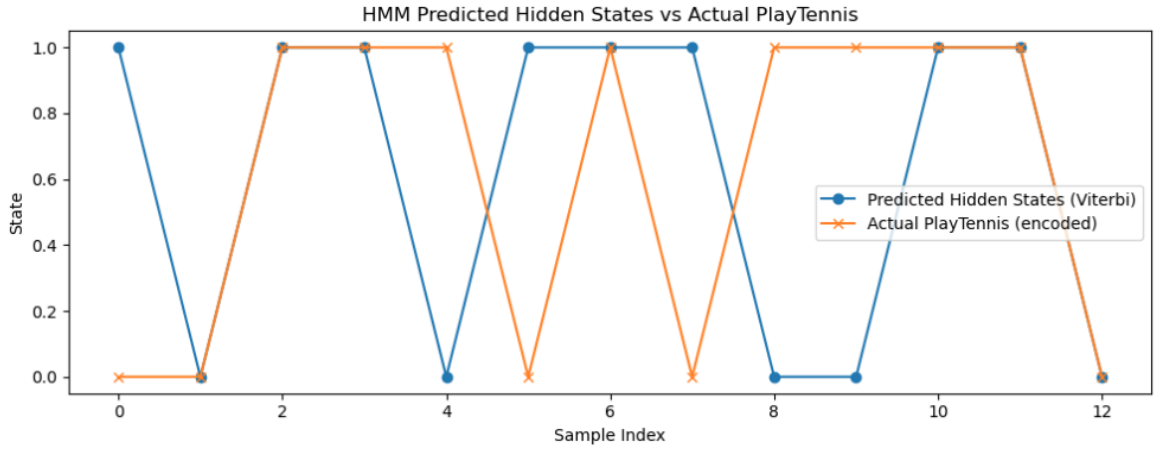
```
    alpha = np.zeros((T, N), dtype=float)
    scale = np.zeros(T, dtype=float)

    # t = 0
    alpha[0, :] = start_p * emit_p[:, obs_idx[0]]
    scale[0] = alpha[0, :].sum()
    if scale[0] == 0:
        scale[0] = 1e-12
    alpha[0, :] /= scale[0]

    for t in range(1, T):
        for j in range(N):
            # sum over previous states i
            alpha[t, j] = emit_p[j, obs_idx[t]] * np.dot(alpha[t - 1, :], trans_p[:, j])
        scale[t] = alpha[t, :].sum()
        if scale[t] == 0:
            scale[t] = 1e-12
        alpha[t, :] /= scale[t]

    # log-likelihood of the sequence under the model
    log_prob = np.sum(np.log(scale))
    return alpha, log_prob

# --- Viterbi algorithm in log-space for numerical stability ---
def viterbi_log(obs_idx, start_p, trans_p, emit_p):
    T = len(obs_idx)
    N = len(start_p)
    log_start = np.log(start_p + 1e-12)
    log_trans = np.log(trans_p + 1e-12)
```

```
    log_emit = np.log(emit_p + 1e-12)

    v = np.zeros((T, N), dtype=float)  # viterbi scores (log probs)
    backpointer = np.zeros((T, N), dtype=int)

    v[0, :] = log_start + log_emit[:, obs_idx[0]]
for t in range(1, T):
        for j in range(N):
            scores = v[t - 1, :] + log_trans[:, j] + log_emit[j, obs_idx[t]]
            backpointer[t, j] = np.argmax(scores)
            v[t, j] = np.max(scores)

    best_last = np.argmax(v[-1, :])
    best_path = np.zeros(T, dtype=int)
    best_path[-1] = best_last
    for t in range(T - 1, 0, -1):
        best_path[t - 1] = backpointer[t, best_path[t]]

    best_log_prob = v[-1, best_last]
    return best_path, best_log_prob

# --- Run algorithms ---
alpha, log_likelihood = forward_scaled(obs_seq_idx, start_prob, trans_prob, emit_prob)
print("Log-likelihood (scaled forward):", log_likelihood)

best_path, best_log_prob = viterbi_log(obs_seq_idx, start_prob, trans_prob, emit_prob)
print("Viterbi best path (state indices):", best_path)
print("Viterbi best path log-prob:", best_log_prob)
```

```
    # Compare to actual labels
    print("Actual PlayTennis (encoded):", y)
    # show readable labels
    try:
        readable = encoders["PlayTennis"].inverse_transform(y)
        print("Actual PlayTennis (original labels):", list(readable))
    except Exception:
        print("Could not inverse-transform PlayTennis labels (they are encoded integers)")

    # --- Plot results ---
    plt.figure(figsize=(10, 4))
    plt.plot(best_path, marker='o', label='Predicted Hidden States (Viterbi)')
    plt.plot(y, marker='x', label='Actual PlayTennis (encoded)')
    plt.xlabel('Sample Index')
    plt.ylabel('State')
    plt.title('HMM Predicted Hidden States vs Actual PlayTennis')
    plt.legend()
    plt.tight_layout()
    plt.show()
except Exception as exc:
    print("An error occurred while running the HMM script:\n")
    traceback.print_exc()
    raise
```

Output:

```
Loaded sample_data.csv with shape: (13, 5)
Number of unique observations: 13
Sequence length: 13, Hidden states: 2
Log-likelihood (scaled forward): -35.13391486826859
Viterbi best path (state indices): [1 0 1 1 0 1 1 1 0 0 1 1 0]
Viterbi best path log-prob: -39.36672352314845
Actual PlayTennis (encoded): [0 0 1 1 1 0 1 0 1 1 1 1 0]
Actual PlayTennis (original labels): ['No', 'No', 'Yes', 'Yes', 'Yes', 'No', 'Yes', 'No', 'Yes', 'Yes', 'Yes', 'Yes', 'No']
```



Learnings:

HMMs are useful when data depends on **sequential patterns** (speech, stock market, text).

hmmlearn simplifies implementation (no need to code algorithms like Forward-Backward or Viterbi manually).

By training on observed data, HMMs can **discover hidden patterns** and classify unseen sequences.

Practical applications include **speech recognition, bioinformatics (gene prediction), POS tagging, and activity recognition**.