Double-click (or enter) to edit

Double-click (or enter) to edit

Markov Model forecast the value of a variable whose predicted value is influenced only by its current state, and not by any prior activity

Markov models are useful when a decision problem involves risk that is continuous over time, when the timing of events is important, and when important events may happen more than once.

Markov Models and Hidden Markov Models are both types of probabilistic models used to represent sequential data, but they differ in the nature of the underlying states and observations.

A Markov Model, also known as a Visible Markov Model, is a statistical model that assumes the probability of each state depends only on the current state and not on the previous states. In a Markov Model, the hidden state sequence is directly observable, and the transitions between states are described by a set of transition probabilities.

On the other hand, a Hidden Markov Model (HMM) is a more complex model where the underlying state sequence is not directly observable. In an HMM, there are two sets of states: the hidden states, which represent the true but unobserved states of the system, and the observed states, which are the outputs or observations that are actually seen. The hidden states are connected by transition probabilities, and each hidden state is associated with a set of observation probabilities that describe the likelihood of observing a particular output

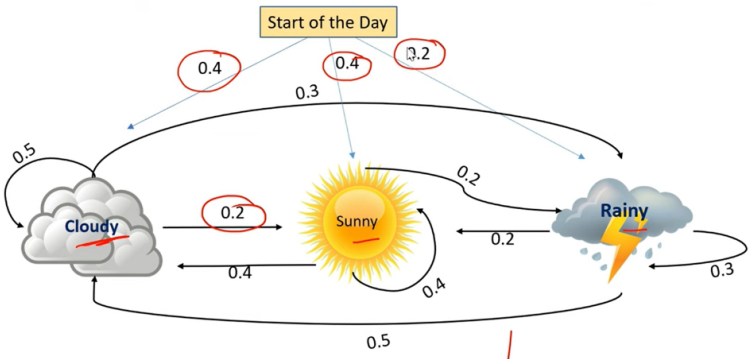
The key differences between Markov Models and Hidden Markov Models are:

Observability of states: In a Markov Model, the states are directly observable, while in a Hidden Markov Model, the states are hidden and must be inferred from the observations.

Complexity: Hidden Markov Models are generally more complex and require more parameters to be estimated, as they involve both the transition probabilities and the observation probabilities.

Applications: Markov Models are often used in simpler, more straightforward applications, such as modeling the weather or user behavior. Hidden Markov Models, on the other hand, are more commonly used in complex applications where the underlying states are not directly observable, such as speech recognition, bioinformatics, and natural language processing.

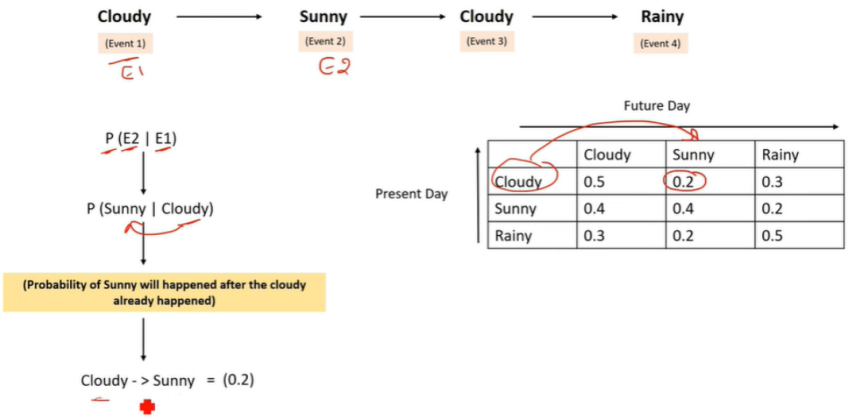
A Hidden Markov Model (HMM) is a statistical model that uses observed parameters to identify hidden parameters, allowing for further analysis



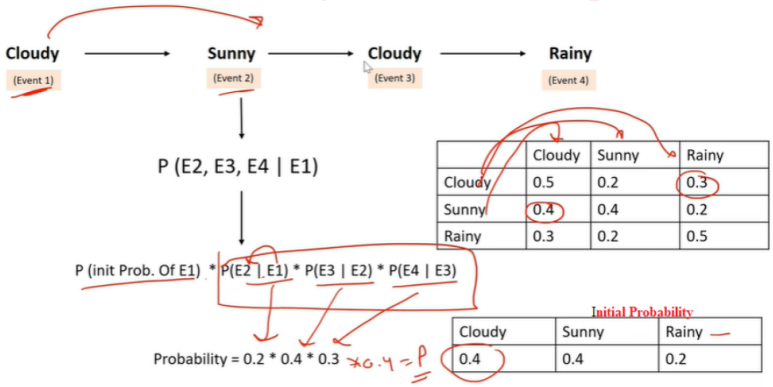
Future Day

Present Day

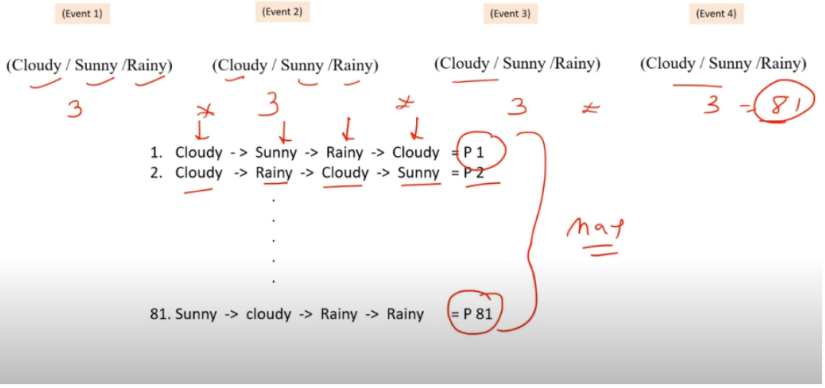
	Cloudy	Sunny	Rainy
Cloudy	0.5	0.2	0.3
Sunny	0.4	0.4	0.2
Rainy	0.3	0.2	0.5



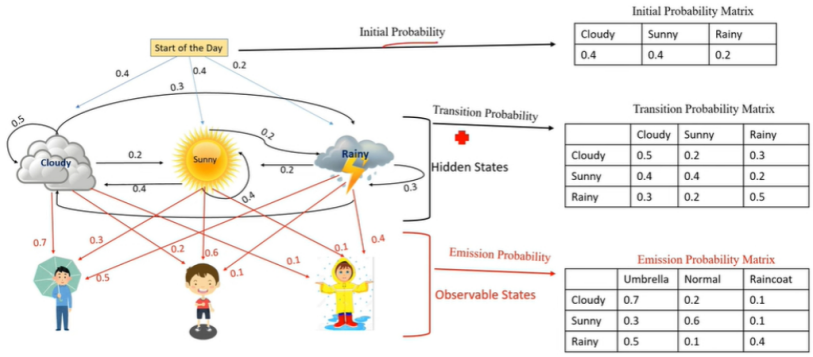
What will be the Probabilty to come Below Sequences



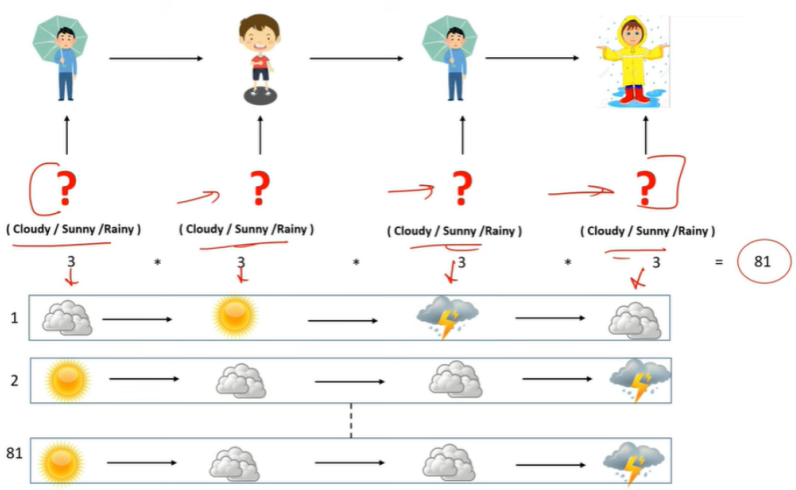
What wll be the in the next four day if event given

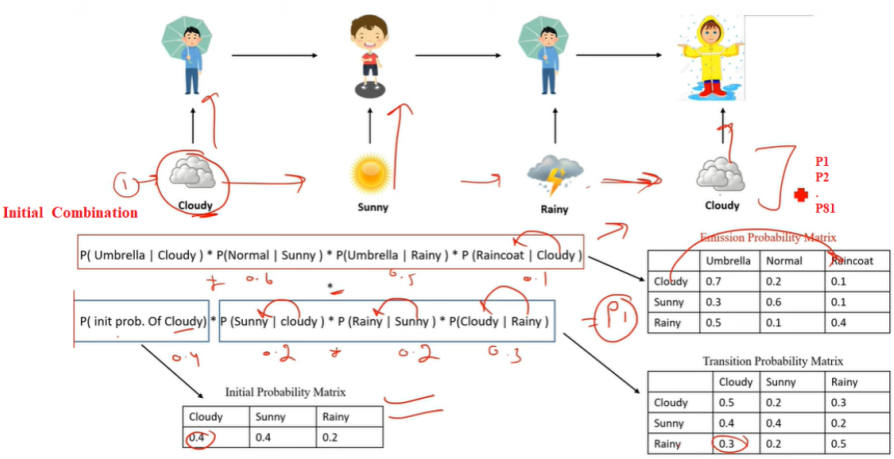


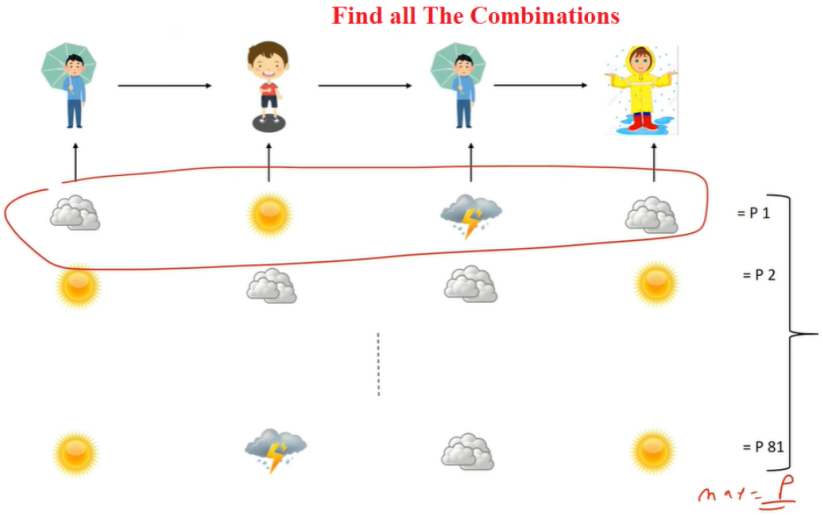
Hidden Markov Model



This is Observable State You have to find hidden state







- HMM can be used in the following applications:

Computational finance speed analysis Speech recognition Speech synthesis Part-of-speech tagging Document separation in scanning solutions Machine translation Handwriting recognition Time series analysis Activity recognition Sequence classification Transportation forecasting

Double-click (or enter) to edit

Double-click (or enter) to edit

Double-click (or enter) to edit

A Markov Model is a probabilistic model that predicts future states based on the current state, using transition probabilities. In Python, you can build a basic Markov Model using dictionaries to represent the state transitions and probabilities.

Let's create a simple example of a Markov Model that predicts the next word in a sequence of text. Here's a step-by-step guide:

Step 1: Prepare the Data

First, we need a sample text or sequence of data from which to learn the transitions.

Step 2: Build the Transition Matrix

We'll use a dictionary where each key is a state (word), and the value is another dictionary representing possible next states and their counts.

Step 3: Convert Counts to Probabilities

Normalize the counts into probabilities to form the Markov Model.

Step 4: Generate Text Using the Markov Model

Use the model to generate new sequences by choosing the next state based on current state probabilities.

Sample text to build the Markov Model

```
import random
text = "the quick brown fox jumps over the lazy dog the quick blue hare jumps over the sleepy dog"
```

Step 1: Split text into words

```
words = text.split()
```

Step 2: Build the transition matrix

```
transition_matrix = {}
```

```
for i in range(len(words) - 1):
    current_word = words[i]
    next_word = words[i + 1]

    if current_word not in transition_matrix:
        transition_matrix[current_word] = {}

    if next_word not in transition_matrix[current_word]:
        transition_matrix[current_word][next_word] = 0

    transition_matrix[current_word][next_word] += 1
```

Step 3: Convert counts to probabilities

```
for current_word, transitions in transition_matrix.items():
    total_count = sum(transitions.values())
    for next_word in transitions:
        transitions[next_word] /= total_count
```

Display the Markov Model

```
print("Markov Model (Transition Matrix):")
for state, transitions in transition_matrix.items():
    print(f"{state}: {transitions}")
```

```
Markov Model (Transition Matrix):
the: {'quick': 0.5, 'lazy': 0.25, 'sleepy': 0.25}
quick: {'brown': 0.5, 'blue': 0.5}
brown: {'fox': 1.0}
fox: {'jumps': 1.0}
jumps: {'over': 1.0}
over: {'the': 1.0}
lazy: {'dog': 1.0}
dog: {'the': 1.0}
blue: {'hare': 1.0}
hare: {'jumps': 1.0}
sleepy: {'dog': 1.0}
```

Step 4: Generate text using the Markov Model

```
def generate_text(model, start_word, length=10):
    current_word = start_word
    result = [current_word]

    for _ in range(length - 1):
        if current_word in model:
            next_words = list(model[current_word].keys())
            probabilities = list(model[current_word].values())
            current_word = random.choices(next_words, probabilities)[0]
            result.append(current_word)
        else:
            break  # Stop if no transitions are available

    return ' '.join(result)
```

Example usage: Generate text starting with 'the'

```
generated_text = generate_text(transition_matrix, start_word='the', length=9)
print("\nGenerated Text:")
print(generated_text)
```

```

Generated Text:
the quick brown fox jumps over the sleepy dog the
```

Explanation

Building the Transition Matrix: We count occurrences of each transition from one word to the next and store them in a dictionary.

Converting to Probabilities: The counts are converted to probabilities by dividing each count by the total counts for the current word.

Generating Text: The model generates text by starting with a specified word and choosing subsequent words based on the transition probabilities.

```
import json
import numpy as np
import pandas as pd
import tensorflow as tf
import json
import numpy as np
import pandas as pd
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.decomposition import PCA
from sklearn.cluster import KMeans
import matplotlib.pyplot as plt
```

```
df = pd.read_csv('https://raw.githubusercontent.com/makhan010385/Mtech-III-Executive--NLP/main/Scrap600
```

```
import pandas as pd
from sklearn.model_selection import train_test_split
```

```
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.preprocessing import OneHotEncoder
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import accuracy_score, classification_report, confusion_matrix
```

```
tfidf = TfidfVectorizer()
statement_features = tfidf.fit_transform(df['statement'])
```

```
Start coding or generate with AI.
```

```
# Combining TF-IDF features with the encoded categorical features
features = scipy.sparse.hstack([statement_features, categorical_features])
```

```
import scipy
features = scipy.sparse.hstack([statement_features, categorical_features])
```

```
# Convert the sparse matrix to a dense array
features = features.toarray()
```

```
# Splitting the data into training and testing sets
X_train, X_test, Y_train, Y_test = train_test_split(features, df['BinaryNumTarget'], test_size=0.3, ran
```

```
# HMM initialization
# n_components: Number of hidden states (e.g., 2 states: fake and real)
model = hmm.GaussianHMM(n_components=2, covariance_type='diag', n_iter=100)
```

```
# Training HMM
model.fit(X_train)
```

```
GaussianHMM
GaussianHMM(n_components=2, n_iter=100)
```

```
# Predicting hidden states (fake or real) on the test set
Y_pred = model.predict(X_test)
```

```
# Mapping HMM states to actual labels
# Here we need to map HMM states to 0 (real) or 1 (fake) based on the majority vote
# You might need to adjust this mapping based on your dataset
state_mapping = {0: 0, 1: 1}  # Example mapping (adjust as needed based on the data)
Y_pred_mapped = np.array([state_mapping[state] for state in Y_pred])
```

```
# Evaluating the model
print("Accuracy:", accuracy_score(Y_test, Y_pred_mapped))
print("Classification Report:\n", classification_report(Y_test, Y_pred_mapped))
```

```
Accuracy: 0.4186307519640853
Classification Report:
              precision    recall  f1-score   support

           0       0.47      0.39      0.43       988
           1       0.37      0.45      0.41       794

    accuracy                           0.42      1782
   macro avg       0.42      0.42      0.42      1782
weighted avg       0.43      0.42      0.42      1782
```

Conversion to Dense Format: features = features.toarray() converts the combined sparse matrix to a dense format compatible with the HMM model.

Splitting Data: Make sure to consistently use train_test_split with appropriate parameters (test_size, random_state) to ensure reproducibility.

State Mapping: Adjust the state_mapping if the HMM states do not align correctly with the real/fake labels. Inspect the predicted states (Y_pred) to correctly map them to your target labels.

```
import numpy as np
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.preprocessing import OneHotEncoder
from sklearn.model_selection import train_test_split, GridSearchCV
from sklearn.linear_model import LogisticRegression
from sklearn.pipeline import make_pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import accuracy_score, classification_report
```

```
from sklearn.ensemble import VotingClassifier
from hmmlearn import hmm
import scipy
```

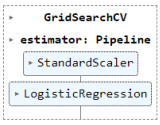
```
# Step 1: Extract TF-IDF features from the statements, including bigrams for more context
tfidf = TfidfVectorizer(ngram_range=(1, 2), max_features=10000)  # Adjust max_features based on availab
statement_features = tfidf.fit_transform(df['statement'])
```

```
# Step 2: One-hot encode the categorical features
encoder = OneHotEncoder(handle_unknown='ignore')
categorical_features = encoder.fit_transform(df[['author', 'source']])
```

```
# Step 3: Combine TF-IDF features with the encoded categorical features
features = scipy.sparse.hstack([statement_features, categorical_features])

# Convert sparse matrix to dense (only for HMM, Logistic Regression can handle sparse matrices)
features_dense = features.toarray()
```

```
# Step 4: Split the data into training and testing sets
X_train, X_test, Y_train, Y_test = train_test_split(features, df['BinaryNumTarget'], test_size=0.3, ran
# Step 5: Initialize and tune Logistic Regression model using Grid Search
logistic_pipeline = make_pipeline(StandardScaler(with_mean=False), LogisticRegression(max_iter=1000, ra
param_grid = {'logisticregression__C': [0.01, 0.1, 1, 10, 100]}  # Adjust C values as needed
logistic_grid = GridSearchCV(logistic_pipeline, param_grid, cv=5, scoring='accuracy')
logistic_grid.fit(X_train, Y_train)
```




```
# Best logistic regression model
logistic_model = logistic_grid.best_estimator_
logistic_probs = logistic_model.predict_proba(X_test)[:, 1]
```

```

# Step 6: Initialize and tune Hidden Markov Model (HMM)
hmm_model = hmm.GaussianHMM(n_components=2, covariance_type='diag', n_iter=200)  # Increase n_iter if n
hmm_model.fit(X_train.toarray())

# Predict hidden states with HMM
hmm_states = hmm_model.predict(X_test.toarray())
```

```
WARNING:hmmlearn.base:Model is not converging.  Current: 171383424.46485186 is not greater than 171383424.46485513. Delta is -3.2782554626464844e-06
```

```
# Convert HMM states to binary predictions (map states directly)
hmm_probs = (hmm_states == 1).astype(int)

# Step 7: Combine predictions using a weighted average (assign higher weight to better-performing model
logistic_weight = 0.7  # Weight for logistic regression
hmm_weight = 0.3       # Weight for HMM
ensemble_probs = (logistic_weight * logistic_probs + hmm_weight * hmm_probs) / (logistic_weight + hmm_w
ensemble_preds = (ensemble_probs > 0.5).astype(int)  # Threshold at 0.5 for binary classification
```

```
# Step 8: Evaluate the ensemble model
print("Accuracy:", accuracy_score(Y_test, ensemble_preds))
print("Classification Report:\n", classification_report(Y_test, ensemble_preds))
```

```
Accuracy: 0.6054994388327721
Classification Report:
              precision    recall  f1-score   support

           0       0.64      0.67      0.65       988
           1       0.56      0.53      0.54       794

    accuracy                           0.61      1782
   macro avg       0.60      0.60      0.60      1782
weighted avg       0.60      0.61      0.60      1782
```