env_sample_generation.py

`add_noise(state, terminated, bound)`

This function adds noise to a state to simulate environmental variability. It ensures the noisy state remains within defined bounds:

Parameters:

- `state`: Current state of the environment.
- `terminated`: Boolean indicating if the episode has terminated.
- `bound`: List of bounds for each state variable.

Returns:

- `noisy_state`: The noisy version of the input state.

`generate_episodes(num_samples, max_episode_length, bound)`

This function generates a specified number of episodes, each consisting of state transitions, actions, and rewards:

Parameters:

- `num_samples`: Number of samples to generate.
- `max_episode_length`: Maximum length of each episode.
- `bound`: List of bounds for each state variable.

Returns:

- `episode_inputs`: Tensor containing input states and actions.
- `episode_outputs`: Tensor containing next states, rewards, and termination flags.
- `df`: DataFrame containing the collected samples.

Saving Data

- **Training and Test Datasets**: Saved as pickle files for later use.
- **Samples DataFrame**: Saved as a CSV file for analysis.

Files Saved:

- `env_name_k_train_dataset.pkl`: Training dataset for the environment.
- `env_name_k_test_dataset.pkl`: Testing dataset for the environment.
- `env_name_k_sample.csv`: CSV file containing raw samples data.

---

state_var_graph.py

For a particular environment, this notebook plots an individual graph of state variable vs time for each model.

---

create_models.py

This notebook trains PETS, Monte Carlo Dropout and Bayesian neural network models.

1. Gaussian and Ensemble Gaussian Network (PETS)

This code defines two classes for neural networks used in probabilistic modeling: `GaussianNetwork` and `EnsembleGaussianNetwork`. Both classes use PyTorch and are designed for tasks where the output is a Gaussian distribution characterized by its mean and standard deviation.

GaussianNetwork Class

Overview

`GaussianNetwork` is a neural network that outputs the parameters of a Gaussian distribution (mean and standard deviation) given an input tensor.

Parameters:

- `input_dim` (int): The dimension of the input features.

- `hidden_dim` (int): The dimension of the hidden layers.
- `output_dim` (int): The dimension of the output features (mean and standard deviation).

Forward Method:

Parameters:

- `x` (torch.Tensor): Input tensor.

Returns:

- `mean` (torch.Tensor): The predicted mean of the Gaussian distribution.
- `std` (torch.Tensor): The predicted standard deviation of the Gaussian distribution, clamped to ensure numerical stability.

EnsembleGaussianNetwork Class

Overview

`EnsembleGaussianNetwork` is a neural network ensemble consisting of multiple `GaussianNetwork` instances. It combines the outputs of these models to provide an ensemble prediction.

Parameters:

- `num_models` (int): The number of `GaussianNetwork` models in the ensemble.
- `input_dim` (int): The dimension of the input features.
- `hidden_dim` (int): The dimension of the hidden layers.
- `output_dim` (int): The dimension of the output features (mean and standard deviation).

Forward Method

Parameters:

- `x` (torch.Tensor): Input tensor.

Returns:

- `means` (torch.Tensor): Stacked means from all models in the ensemble.
- `stds` (torch.Tensor): Stacked standard deviations from all models in the ensemble.

2. MC_Dropout_Net Class

Parameters

- **input_size** (int): The size of the input layer.
- **output_size** (int): The size of the output layer.
- **num_hidden_layers** (int): The number of hidden layers in the network.
- **hidden_layer_nodes** (int): The number of nodes in each hidden layer.
- **activation** (nn.Module): The activation function to be used in the hidden layers.
- **dropout_prob** (float): The dropout probability for regularization.
- **num_network** (int): The number of sub-networks to be used for Monte Carlo Dropout.

3. BayesianNN Class

Parameters

- **input_size** (int): The size of the input layer.
- **output_size** (int): The size of the output layer.
- **num_hidden_layers** (int): The number of hidden layers in the network.
- **hidden_layer_nodes** (int): The number of nodes in each hidden layer.
- **activation** (nn.Module): The activation function to be used in the hidden layers.

---

env_sample_generation.py

Sample Class

The `Sample` class generates samples from a given model and environment, storing them in a DataFrame and saving them to a CSV file.

Parameters

- **model** (nn.Module): The neural network model used to predict the next state given the current state and action.
- **env** (gym.Env): The OpenAI Gym environment to interact with.

- **df_data** (pd.DataFrame): A DataFrame to store the generated samples.

Methods

`__init__`

The constructor method initializes the `Sample` class with the specified model and environment.

`generateSample`

The method generates samples from the environment using the model, saves the samples in a DataFrame, and writes the DataFrame to a CSV file.

Parameters

- **filename** (str): The name of the file to save the generated samples.
- **num_samples** (int): The number of samples to generate.
- **max_episode_length** (int, optional): The maximum length of an episode. Defaults to 200.
- **noise** (float, optional): The noise to add to the next state. Defaults to 0.

---

mdp_construction_all_model_samples.py

MDP Class

Overview

The `MDP` class is designed to handle Markov Decision Processes (MDPs) by processing samples from various models, discretizing variables, constructing the MDPs, and providing visualization and comparison capabilities.

Class: `MDP`

`__init__(self, env_name, num_samples, action_discretize_bins=None, action_bound=None)`

Parameters:

- `env_name` (str): Name of the environment.
- `num_samples` (str): Number of samples to be processed.
- `action_discretize_bins` (int, optional): Number of bins for action discretization. Default is `None`.
- `action_bound` (list, optional): Bounds for action discretization. Default is `None`.

Description:

Initializes the MDP object by loading sample data for the original, PETS, Bayesian, and Monte Carlo Dropout models. It optionally discretizes actions if `action_discretize_bins` and `action_bound` are provided.

`discretize(self, bound, no_of_bins, include_out_of_bounds)`

Parameters:

- `bound` (list): Bounds for discretizing variables.
- `no_of_bins` (list): Number of bins for each variable.
- `include_out_of_bounds` (bool): Whether to include out-of-bound values in the discretization.

Description:

Discretizes the variables in the sample data according to the provided bounds and number of bins. It handles both within-bounds and out-of-bounds cases.

`discretize_variables(self, df, variable, nx_variable, bound, no_of_bins)`

Parameters:

- `df` (pd.DataFrame): DataFrame containing the data to be discretized.
- `variable` (str): Name of the variable to be discretized.
- `nx_variable` (str): Name of the next variable to be discretized.
- `bound` (list): Bounds for discretization.
- `no_of_bins` (int): Number of bins for discretization.

Description:

Discretizes a variable within specified bounds and number of bins. Optionally discretizes the next variable if provided.

`discretize_variables_within_bounds(self, df, variable, nx_variable, bound, no_of_bins)`

Parameters:

- `df` (pd.DataFrame): DataFrame containing the data to be discretized.
- `variable` (str): Name of the variable to be discretized.
- `nx_variable` (str): Name of the next variable to be discretized.
- `bound` (list): Bounds for discretization.
- `no_of_bins` (int): Number of bins for discretization.

Description:

Discretizes variables strictly within the provided bounds and number of bins. Removes rows with out-of-bound values.

`mdp_construct(self, bound, no_of_bins, include)`

Parameters:

- `bound` (list): Bounds for discretizing variables.
- `no_of_bins` (list): Number of bins for each variable.
- `include` (bool): Whether to include out-of-bound values in the discretization.

Description:

Constructs the MDP by discretizing variables, computing transition probabilities, and saving the results. It also generates visualizations of the constructed MDPs.

`prob_construct(self, df)`

Parameters:

- `df` (pd.DataFrame): DataFrame containing discretized data.

Description:

Constructs the probability transition matrix for the given discretized data.

`comparison(self, no_of_bins, include)`

Parameters:

- `no_of_bins` (int): Number of bins used for discretization.

- `include` (bool): Whether out-of-bound values were included in the discretization.

Description:

Compares the sizes of the constructed MDPs and calculates the KL divergence between different models. Saves the comparison results to a file.

`mdp_size(self, df)`

Parameters:

- `df` (pd.DataFrame): DataFrame containing the MDP data.

Description:

Calculates and returns the number of unique states in the MDP.

`kl_divergence(self, df1, df2)`

Parameters:

- `df1` (pd.DataFrame): First DataFrame containing MDP data.
- `df2` (pd.DataFrame): Second DataFrame containing MDP data.

Description:

Calculates and returns the KL divergence between the two MDPs.

`visualize_discreteMDP_networkx(self, df=None, save_path=None)`

Parameters:

- `df` (pd.DataFrame, optional): DataFrame containing the MDP data to be visualized. Default is `None`.
- `save_path` (str, optional): Path to save the visualization. Default is `None`.

Description:

Visualizes the MDP using NetworkX. If a save path is provided, the visualization is saved as a PNG file; otherwise, it is displayed.

---

onnx_to_constraints.py

ONNX to Constraints Conversion

This notebook demonstrates the process of converting an ONNX model to constraints, specifically for use with constraint solvers like Z3. The steps involve loading the ONNX model, extracting weights, marking specific activation layers, creating Z3 variables, and generating the corresponding SMT constraints.

Steps

1. **Load ONNX Model and Weights:**
    - The code loads the ONNX model and extracts the weights from its initializers.
2. **Extract Model Information:**
    - It iterates through the graph of the ONNX model to identify the activation functions used (ReLU in this case) and the number of hidden layers.
3. **Create SMT Variables:**
    - The code creates Z3 solver variables for the model's inputs, outputs, hidden layers, and activation layers.
4. **Build SMT Constraints:**
    - It builds SMT constraints for each layer of the Keras model, representing the mathematical operations between weights, biases, and activations.
    - It also adds constraints to ensure the input values fall within the range specified in the Excel sheet.
5. **Solve SMT Constraints:**
    - The code attempts to solve the SMT constraints using a Z3 solver.
    - If a solution is found, it compares the predicted outputs from the SMT solver with the actual outputs from the Keras model (loaded from a separate ONNX session).
6. **Convert ONNX Model to Keras Model:**
    - Finally, the code converts the original ONNX model to a Keras model, potentially modifying node and input names to avoid naming conflicts.

Key Functions

- `get_hidden_units(weights)`: This function calculates the number of hidden units in each layer based on the weight shapes.
- `smt_min_max(val)`: This function converts range values (from the Excel sheet) into a format suitable for SMT constraints.
- `onnx_to_keras(onnx_model, input_names)`: This function (likely from an external library) converts the ONNX model to a Keras model.

DQN-cartpole.py

Documentation for DQN Training Code

This code implements a Deep Q-Network (DQN) agent for training on a reinforcement learning environment.

Functions:

- **build_model(input_dim, output_dim):**
    - This function creates a sequential neural network model for the DQN agent.
    - **Parameters:**
        - `input_dim`: Integer representing the dimensionality of the environment's state space.
        - `output_dim`: Integer representing the number of actions available in the environment.
    - **Returns:**
        - A compiled Keras model ready for training.
- **train_dqn(env, episodes=1000, epsilon_decay=0.995):**
    - This function trains the DQN agent on a given environment.
    - **Parameters:**
        - `env`: A Gym environment object representing the environment to interact with.
        - `episodes` (Optional): Integer specifying the number of training episodes (default: 1000).
        - `epsilon_decay` (Optional): Float value between 0 and 1 controlling the decay rate of exploration (default: 0.995).
    - **Returns:**
        - A tuple containing the trained DQN model and the replay memory buffer.

Training Process:

1. **Model Building:**
    - The `build_model` function creates a simple DQN model with two hidden layers of 24 units each with ReLU activation.
    - The output layer has a linear activation and its size matches the number of available actions in the environment.
2. **Training Loop:**

- The `train_dqn` function iterates for a specified number of episodes.
- Inside each episode:
  - The environment is reset, and the initial state is obtained.
  - An episode length counter and total reward accumulator are initialized.
  - While the episode is not done or the maximum episode length is not reached:
    - An exploration-exploitation trade-off is made using an epsilon-greedy policy.
    - With probability `epsilon`, a random action is chosen.
    - Otherwise, the action with the highest predicted Q-value from the model is selected.
    - The action is taken in the environment, resulting in a new state, reward, and "done" flag.
    - The experience (state, action, reward, next_state, done) is stored in a replay memory buffer.
    - The state is updated to the next state, and the reward is accumulated.
  - Once the episode terminates, experience replay is performed:
    - If there are enough experiences in memory (greater than batch size), a random batch is sampled.
    - For each experience in the batch:
      - A target Q-value is calculated based on the reward and the discounted maximum Q-value of the next state (Bellman equation).
      - The model is updated to predict the target Q-value for the chosen action in the current state.
  - The exploration rate (`epsilon`) is decayed for the next episode.
  - The total episode reward is recorded.

3. **Evaluation:**
   - After training, the function plots a graph depicting the total reward achieved across episodes to visualize the learning progress.

---

pets_example_mbrllib.py

Use our mbrl toolbox to write the PETS algorithm, and use it to solve a continuous version of the cartpole environment. PETS is a model-based algorithm that consists of two main components:

an ensemble of probabilistic models (each a feed-forward neural network), and a planner using the Cross-Entropy Method .

A basic implementation of this algorithm consists of the following sequence of steps:

1. Gather data using an exploration policy
2. Repeat:
   2.1. Train the dynamics model using all available data.
   2.2. Do a trajectory on the environment, choosing actions with the planner, using the dynamics model to simulate environment transitions.

The ensemble model is trained to predict the environment's dynamics, and the planner tries to find high-reward trajectories over the model dynamics.

To implement this using `MBRL-Lib`, we will use an ensemble of neural networks (NNs) modelling Gaussian distributions (available in the mbrl.models module), and a trajectory optimizer agent that uses CEM (available in the mbrl.planning module). We will also rely on several of the utilities available in the mbrl.util module. Finally, we will wrap the dynamics model into a gym-like environment over which we can plan action sequences.

---

env_discretisedMDP.py

Documentation for MDP Class

This document describes the `MDP` class used to process data from a Gym environment and construct a Markov Decision Process (MDP) representation.

Class Attributes:

- `env`: Gym environment object.

Data Attributes:

- `df_discrete` (pandas.DataFrame): DataFrame containing discretized samples for MDP construction. (Populated by `generateSample` and `discretizeFromJSON`)
- `df_MDP` (pandas.DataFrame): DataFrame containing the constructed MDP with transition probabilities. (Populated by `constructMDP` or `constructMDPforDF`)

Methods:

- **`__init__(self, env)`**
    - Initializes the class with a Gym environment object.
- **`generateSample(self, episodes=1000, max_episode_length=2000, noise=0)`**
    - Generates sample data from the Gym environment.
        - `episodes`: Number of episodes to simulate (default: 1000).
        - `max_episode_length`: Maximum number of steps per episode (default: 2000).
        - `noise`: Amount of noise to add to next state (default: 0).
    - Stores the generated samples in `self.df_data` as a pandas DataFrame.
- **`discretizeFromBins(self, bins)`**
    - Discretizes the state and next state features in the samples DataFrame (`self.df_data`) based on provided bins.
        - `bins`: List containing the number of bins for each state feature (length should match state dimension).
    - Creates new columns with discretized states and next states.
- **`discretizeFromJSON(self, jsonFile)`**
    - Discretizes the state and next state features in the samples DataFrame (`self.df_data`) based on a JSON file definition.
    - The JSON file should specify the column names, binning method (number of bins or bin size), and optionally a list of bins.
    - Creates new columns with discretized states and next states.
- **`constructMDPfromDiscreteFeature(self, output_file=None)`**
    - Constructs the MDP from the discretized samples (`self.df_discrete`).
    - Calculates transition probabilities between states and actions.
    - Stores the MDP as a DataFrame in `self.discrete_MDP`.
    - Optionally saves the MDP to an excel file (`output_file`).
- **`constructMDP(self, output_file=None)`**
    - Constructs the MDP from the original state features (not discretized) in `self.df_data`.
    - Calculates transition probabilities between states and actions.
    - Stores the MDP as a DataFrame in `self.df_MDP`.
    - Optionally saves the MDP to an excel file (`output_file`).

- **`constructMDPforDF(self, df, output_file)`** (helper function)
    - Constructs the MDP from a provided pandas DataFrame (`df`).
    - Calculates transition probabilities between states and actions.
    - Returns the MDP as a DataFrame.
    - Optionally saves the MDP to an excel file (`output_file`).
- **`min_max_states(self, non_null_values_col, non_null_values_col_nx)`** (helper function)
    - Finds the minimum and maximum values between two data columns.
    - Used for determining bin ranges during discretization.
- **`naming_state_col(self, col_name)`** (helper function)
    - Creates a consistent naming convention for state and next state columns based on a provided column name.
- **`verification(self)`**
    - Prints the contents of `self.discrete_MDP` for verification purposes.
- **`visualize_MDP_networkx(self, save_path=None)`**
    - Visualizes the MDP as a directed graph using NetworkX.
    - Optionally saves the visualization to an image file (`save_path`).
- **`add_noise(self, state)`**
    - Adds noise to the next state value.
    - You can adjust the noise level by modifying the `scale` parameter in the normal distribution.
- **`visualize_discreteMDP_networkx(self, df=None, save_path=None)`**
    - Visualizes the discretized MDP as a directed graph using NetworkX.
        - `df`: Optional DataFrame to use for visualization (defaults to `self.discrete_MDP`).
    - Optionally saves the visualization to an image file (`save_path).

---

graphviz_plot_mdp.py

MDP Data Visualization Tool

This script generates a visual representation of a Markov Decision Process (MDP) from data stored in an Excel file.

Input:

The script expects an Excel file named `mdp_2d_data.xls` containing data that defines the MDP. The data should be organized as follows:

- Columns starting with `nx_` represent state variables of the MDP.
- A column named `action` specifies the action taken in each transition.
- A column named `probability` (optional) can specify the probability of each transition (not used in this script).
- A column named `reward` (optional) can specify the reward associated with each transition (not currently visualized).

Output:

The script generates a file named `mdp.dot` containing Graphviz code that represents the MDP as a directed graph. This code can be used with Graphviz software (e.g., `dot` command) to visualize the MDP.

Process:

1. **Import libraries:** The script imports the pandas library for data manipulation.
2. **Read data:** The script reads data from the `mdp_2d_data.xls` file and stores it in a pandas dataframe.
3. **Extract state variables:** The script identifies the state variables from the dataframe columns starting with `nx_` and removes prefix `nx_` to obtain the actual state names.
4. **Generate Graphviz code:** The script iterates through each row of the data. For each row, it extracts the current state, action, and next state. It then constructs Graphviz code defining a directed edge from the current state to the next state with the action label.
5. **Save Graphviz code:** The generated Graphviz code is saved to a file named `mdp.dot`.

** (Optional) Visualization with Graphviz:**

The script includes commented-out code demonstrating how to use the `dot` command-line tool from Graphviz to generate an image (`.png`) of the MDP based on the created `.dot` file.

networkx_plot_mdp.py

Code Purpose:

This Python code constructs a directed graph representation of a Markov Decision Process (MDP) from data stored in a CSV file (`mdp_2d_data.xls`). The MDP graph visualizes the states, actions, and transitions between them, aiding in understanding and analyzing the decision-making process within the MDP.

Explanation:

1. **Import Libraries:**
    - `pandas (pd)`: Used for data manipulation (reading CSV files, creating DataFrames).
    - `networkx (nx)`: Facilitates graph creation, manipulation, and visualization.
    - `matplotlib.pyplot (plt)`: Enables plotting the MDP graph.
2. **Load MDP Data:**
    - Comments the original `pd.read_excel` line (use it if your data is in an Excel file).
    - Uses `pd.read_csv` to read data from the CSV file `mdp_2d_data.xls`. Assumes the first column contains state names and subsequent columns (starting with `nx_`) represent state features.
3. **Identify State Variables:**
    - Creates a list `nx_state_var` that filters columns starting with `nx_` (excluding `action` and `probability` columns) to identify state feature columns.
    - Creates a list `state_var` by removing the `nx_` prefix from each element in `nx_state_var`. This list stores the actual state variable names.
4. **Create Directed Graph:**
    - Initializes `mdp_graph` as a directed graph using `nx.DiGraph()`.
5. **Add States as Nodes:**
    - Iterates through each row (data point) in the DataFrame using `df.iterrows()`.
    - Extracts the current state features (excluding action and probability) as `state` and converts it to a comma-separated string for unique identification.
    - Adds the state as a node to the `mdp_graph`.
6. **Add Transitions as Edges:**
    - Iterates through each row (data point) in the DataFrame using `df.iterrows()`.

- Extracts the current state features (excluding action and probability) as `state` and converts it to a comma-separated string.
- Extracts the action taken from the current state (`action`).
- Extracts the next state features (`next_state`) and converts it to a comma-separated string.
- Adds a directed edge from the current state (`state`) to the next state (`next_state`) with the action label (`action`) to the graph.

7. **Plot the MDP Graph:**
   - Uses `nx.spring_layout(mdp_graph)` to generate a node layout (you can experiment with different layout algorithms).
   - Employs `nx.draw_networkx` to visualize the graph with node labels enabled (`with_labels=True`).
   - Retrieves edge attributes (action labels) using `nx.get_edge_attributes(mdp_graph, 'action')`.
   - Overlays the action labels on the edges with `nx.draw_networkx_edge_labels`.
   - Displays the final MDP graph using `plt.show()`.

Sampling-from-MDP-single-column.py

1. Data Preparation

- This code snippet assumes a pandas dataframe `df` is available.
- It creates a list `nx_state_var` containing all column names that start with 'nx' and excludes 'action', 'probability' and 'reward' columns.
- It creates another list `state_var` by removing the 'nx_' prefix from each column name in `nx_state_var`.
- It adds two new columns 'current_state' and 'next_state' to the dataframe `df`.
  - 'current_state' is created by joining the values from columns in `state_var` with ', ' as a delimiter.
  - 'next_state' is created similarly by joining the values from columns in `nx_state_var`.
- It extracts all unique values from the 'current_state' column and stores them in the variable `states`.

2. Sample Generation

- It defines the number of samples to generate (`num_samples`).

- It creates an empty list `random_samples` to store the generated samples.
- It iterates for `num_samples` times:
    - It randomly selects a state from the unique states (`random_state`).
    - It randomly selects an action available in that state (`sample_action`).
    - It filters the dataframe to get the row corresponding to the chosen state and action, weighted by the 'probability' column.
    - It extracts the 'current_state', 'action', and 'next_state' values from the sampled row.
    - It appends a list containing these values to the `random_samples` list.
- Finally, it creates a new dataframe `df_samples` from the `random_samples` list.

3. Probability Calculation

- It calculates the empirical transition probabilities by grouping the dataframe `df_samples` by 'current_state', 'action', and 'next_state' and counting the occurrences. This is stored in `df_empirical_probs`.
- It calculates the total number of times each state-action pair appears in `df_samples` and stores it in `df_empirical_count`.
- It merges `df_empirical_probs` and `df_empirical_count` on the columns 'current_state' and 'action'.
- It calculates the empirical probability for each state-action-next_state combination by dividing the count by the total count for that state-action pair and stores it in the new column 'empirical_probability'.

4. Merging and Comparison

- It merges the original dataframe `df` with `df_empirical_probs` on the columns 'current_state', 'action', and 'next_state'.
- It adds a new column 'prob_diff' to the merged dataframe which is the difference between the specified probability (`probability` column) and the empirical probability (`empirical_probability` column).
- It checks if all the specified probabilities exactly match the empirical probabilities using `np.allclose` and stores the result in `all_probs_match`.
- Finally, it prints the merged dataframe and a message indicating whether all the probabilities match.

Sampling-from-MDP-multi-column.py

Documentation for Python code to calculate empirical probabilities

This code takes a DataFrame (`df`) as input, assumed to contain data for a Markov Decision Process (MDP). It calculates the empirical probabilities of state transitions and compares them to the provided probabilities in the DataFrame.

Explanation:

1. **Identifying state features:**
    - Identifies state features (columns) that don't start with 'nx', aren't 'action', 'probability', or 'reward'.
    - Creates a list of 'next state' features by prefixing 'nx_' to the identified state features.
2. **Creating state representations:**
    - Creates 'current_state' and 'next_state' columns in `df` by converting each row in the respective feature sets to lists.
3. **Generating unique states:**
    - Extracts unique states from 'current_state' features.
4. **Generating random samples:**
    - Generates a specified number (`num_samples`) of random samples.
    - For each sample:
        - Selects a random state from the unique states.
        - Merges the selected state with the DataFrame to get the corresponding action and next state based on probabilities.
        - Randomly selects an action based on the provided probabilities in the merged DataFrame.
        - Randomly selects a next state based on the transition probability for the chosen action.
        - Creates a sample containing current state, action, and next state.
    - Stores all samples in a list (`random_samples`).
5. **Creating DataFrame from samples:**
    - Defines column names based on state features, action, and next state features.
    - Creates a DataFrame (`df_samples`) from the list of samples using the defined column names.
6. **Calculating empirical probabilities:**
    - Calculates the number of occurrences of each combination of state-action-next state in `df_samples`.
    - Creates a DataFrame (`df_empirical_probs`) showing these counts and resets the index.
    - Calculates the total count of occurrences for each state-action pair.

    - Creates another DataFrame (`df_empirical_count`) showing these total counts and resets the index.
    - Merges `df_empirical_probs` and `df_empirical_count` on state and action features.
    - Calculates the empirical probability for each state-action-next state combination by dividing the count by the total count for the corresponding state-action pair.
    - Updates `df_empirical_probs` with the calculated empirical probabilities.
7. **Merging results with original DataFrame:**
    - Merges `df` with `df_empirical_probs` on all features (current state, action, and next state).
8. **Comparing empirical vs. specified probabilities:**
    - Creates a new column (`prob_diff`) in the merged DataFrame to show the difference between the specified probability and the calculated empirical probability for each state-action pair.
    - Checks if all specified probabilities exactly match the empirical probabilities.
9. **Output:**
    - Prints the merged DataFrame showing original data, empirical probabilities, and the difference between specified and empirical probabilities.
    - Prints a message indicating whether all specified probabilities match the empirical probabilities.
