Pet-Match Recommender: Project Overview and Architecture

This document provides an overview of the Pet-Match Recommender project, detailing its components, data flow, and current progress. The goal is to build a prototype demonstrating how generative AI and advanced analytics can enhance shelter adoption rates by matching adopters with suitable pets.

I. Data Foundation and Preprocessing

This section outlines the initial data generation, preparation, and the creation of a labeled training dataset.

1. **Adopter Data Generation (`01_make_adopters.ipynb` equivalent):

   - **Description:** A synthetic dataset of adopter personas forms a key input. As outlined in the initial project plan ("Create an adopter table and pseudo-labels"), this step was necessary due to the lack of readily available real-world adopter data for prototype development. The aim was to "invent a small but consistent adopter-profile table" using tools like Faker to sample attributes such as age, housing type, activity level, and prior pet ownership.
   - **Implementation:** The initial generation of this data was performed using scripts and notebooks (e.g., `/Users/jeremy/Downloads/Adopter Dataset Creation.ipynb`), aligning with the project plan's directive to "Build 5–10 adopter personas… save 5,000–10,000 rows."
   - **Key Output:** An "adopters_silver" dataset, stored in Google Cloud Storage (e.g., `gs://pairing-demo-bucket/Adopters/`).

2. **Pet Data Sourcing & Preparation:

   - **Description:** An existing dataset, `pets_silver`, containing information about adoptable pets, is utilized.
   - **Implementation:** Initial loading and schema understanding were part of early data exploration (e.g., in `/Users/jeremy/Downloads/Dataset_Creation_and_Loading.ipynb`).
   - **Key Output:** The `pets_silver` dataset, accessible in Google Cloud Storage (e.g., `gs://pairing-demo-bucket/pets_silver/`).

3. **Training Pair Generation (`02_make_pairs.ipynb` equivalent):

   - **Description:** To create a supervised learning dataset, adopter and pet data are cross-joined. A rule-based scoring function evaluates each potential (adopter, pet) pair. Pairs are then labeled as good (1) or bad (0) matches based on a predefined score threshold, resulting in a balanced training set.
   - **Implementation:** This core logic is encapsulated in the `PairGCPLoad.py` script, which uses pandas to efficiently process the data and interact with Google Cloud Storage. The script represents an operationalized version of pairing logic prototyped in local notebooks (e.g., `/Users/jeremy/Downloads/Dataset_Creation_and_Loading.ipynb`).
   - **Key Output:** The `train_pairs.parquet/` dataset, stored in GCS (e.g., `gs://pairing-demo-bucket/train_pai` This dataset includes engineered features for both adopters and pets, along with the binary match label, and serves as the primary input for the recommendation model.
   - **Noteworthy Milestones & Adaptations during this stage:**
     - Successful configuration of the GCP environment for GCS bucket creation and permission management.
     - Initial attempts with Dataproc Serverless were unsuccessful due to CPU quota issues.
     - **Pivoted to pandas-based implementation:** This involved using pandas with the `gcsfs` package to interact with GCS for I/O, providing a simpler and more efficient solution for this specific task. The script `PairGCPLoad.py` was developed for this.
     - Adaptation of the pairing script to handle data variability, such as the absence of a "Weight (lbs)" column in the `pets_silver` data. This involved removing weight-dependent features and adjusting the scoring logic to neutralize the impact of the missing size component.

II. Recommendation Model (`03_train_wd_model.py`)

This component focuses on the machine learning model responsible for predicting pet-adopter compatibility.

1. **Selected Model Architecture:**
   - The project employs a **Wide-&-Deep** learning model. This architecture is well-suited for recommendation tasks with tabular data, as it combines the strengths of memorization (wide component) and generalization (deep component).
   - **Tooling:** Implemented using TensorFlow Keras. The design aligns with principles of `tf.estimator.DNNLinearCombinedClassifier` or a similar custom Keras structure. The script `03_train_wd_model.py` handles the training.
2. **Data Handling for Training:**
   - The primary input for training is the `train_pairs.parquet/` dataset (located at `gs://pairing-demo-bucket/tra` generated by `PairGCPLoad.py`.
   - This dataset is downloaded locally to `/Users/jeremy/BigDataProj/model_training/data/train_pairs_local/` for training.
   - It contains features and labels, loaded into a Pandas DataFrame, and then converted to `tf.data.Dataset` for efficient training.
3. **Model Structure & Training Process:**
   - **Wide Component:** Processes sparse, often one-hot encoded, features (e.g., pet breed, color, adopter housing) to learn feature interactions directly.
   - **Deep Component:** A Deep Neural Network (DNN) that learns complex patterns from dense numerical features (e.g., pet age, adopter age) and also incorporates embeddings of categorical features.
   - The model is trained using TensorFlow, with data typically fed through efficient input pipelines (e.g., `tf.data.Dataset`). Callbacks for early stopping and model checkpointing (saving the best model based on validation AUC) are used.
4. **Evaluation and Output:**
   - Model performance is assessed on a hold-out portion (20%) of the `train_pairs.parquet/` data.
   - **Achieved Performance:**
     - Test Loss: 0.0824
     - Test Accuracy: 0.9686
     - Test AUC: 0.9962
   - The trained and validated model is serialized and saved as `pet_model.h5` locally in `/Users/jeremy/BigDataProj/model_training/` and then uploaded to `gs://pairing-demo-bucket/models/pet_`
5. **Challenges and Resolutions during Model Training:**
   - **Input Shape Mismatches:** Initial training attempts faced `ValueError` exceptions due to incompatible input shapes between the `tf.data.Dataset` and the Keras model's `Input` layers.
     - The Keras `Input` layers were defined with `shape=(1,)`, expecting 2D tensors like `(batch_size, 1)`.
     - However, the `tf.data.Dataset` was yielding 1D tensors `(batch_size,)` for features.
   - **Normalization Layer `adapt()` Issues:** The `tf.keras.layers.Normalization` layer's `adapt()` method was initially called with 1D data `(num_samples,)`, leading to an incorrect internal state (a very large number of non-trainable parameters was a symptom). It expects data of the same rank it will receive during training, i.e., `(num_samples, 1)` if processing single features.
   - **Resolution:**
     - The data for numerical features passed to `normalizer.adapt()` was reshaped from `(num_samples,)` to `(num_samples, 1)`.
     - In the `prepare_dataset` function, both numerical and categorical feature arrays were reshaped from `(num_samples,)` to `(num_samples, 1)` before being converted into the `tf.data.Dataset`. This ensured consistency in data rank throughout the pipeline.

III. Streamlit Demonstration Application (`app.py`)

An interactive web application built with Streamlit serves as the front-end to showcase the pet recommendation system.

1. **Core Technologies:**
    - Python, Streamlit, TensorFlow, Pandas, `llama-cpp-python` for local LLM explanations.
2. **Key Loaded Resources for the Application:**
    - **Trained Recommendation Model:** The `pet_model.h5` file.
    - **Llama GGUF Model:** Loaded locally for generating match explanations. The path is configurable via the `LLAMA_MODEL_PATH` environment variable (defaults to `streamlit_app/models/llama-3-8b-instruc`
    - **Pet Information:** The `pets_silver` dataset (or a suitable subset) providing details for recommended pets.
    - **Adopter Data (for options):** The `adopters_silver.parquet` data is used to dynamically populate dropdown options for adopter characteristics (e.g., housing type, activity level), ensuring consistency with the training data's vocabulary.
3. **Application Features and Workflow:**
    - **Dynamic Adopter Profile Input:** Users define their profile directly within the app using interactive widgets in the sidebar (`st.sidebar`). These include:
        - `st.number_input` for Age and Household Size.
        - `st.selectbox` for Housing Type and Activity Level (options are dynamically populated from the `adopters_silver` data or use sensible defaults).
        - `st.radio` for indicating prior pet ownership.
    - **Profile Construction:** The application constructs an adopter profile on-the-fly based on these inputs.
    - **Recommendation Generation:** Upon clicking a "Find Matches" button (`st.button`), a function (`rank_pets`) processes this dynamically created adopter profile and the available pet data through the trained model. It generates a ranked list of top-K (e.g., 3) pet recommendations.
    - **Display of Recommendations:** Recommended pets are displayed with relevant details (e.g., Pet ID, Name, Type, Breed, Color, Age, Sex) and their match score (`st.metric`).
    - **LLM-Generated Explanations:** For each match, a brief explanation is generated by a local Llama (instruct-tuned) model, highlighting why the pet might be a good fit for the adopter's profile.
    - **User Feedback Mechanism (Potential Extension):**
        - UI elements (e.g., like/dislike buttons) can be incorporated to gather user feedback on suggestions, which could be used for future system improvements.
    - **Challenges and Iterations during App Development:**
        - **Initial Adopter Persona Selection:** The first version of the app used a dropdown to select from pre-defined adopter personas. This was changed to direct input for greater flexibility.
        - **TensorFlow `dtype` Mismatches:** Several iterations were required to ensure that data passed to the `model.predict()` method had the correct `dtype` (e.g., `tf.string` for categorical features, `tf.int64` for numerical) and shape, resolving errors like "Invalid dtype: object" and "Invalid dtype: str…". This involved explicit conversion to `tf.constant` with specified dtypes in the `preprocess_input_for_model` function.
        - **Streamlit `set_page_config()` Error:** Encountered and resolved `StreamlitSetPageConfigMustBeFirstCom` by ensuring `st.set_page_config()` was the very first Streamlit command executed in the script.

Recent Application Enhancements and Fixes (Late May 2024)

During a focused troubleshooting and enhancement session, several key improvements were made to the Streamlit application:

- **Llama-cpp-python Installation for macOS (Metal Support):** Resolved `ModuleNotFoundError` for `llama_cpp` on an Apple Silicon (M-series) Mac by reinstalling `llama-cpp-python` with specific compilation flags to enable Metal GPU acceleration. This involved using the command: `bash CMAKE_ARGS='-DLLAMA_METAL=on' FORCE_CMAKE=1 pip install -U llama-cpp-python --no-cache-dir`
- **Keras Model Input Correction:** Addressed errors related to `Missing data for input` and discrepancies in expected feature names by the trained Keras model (`pet_model.h5`).
  - The feature names used in the `preprocess_input_for_model` function within `app.py` were updated to match the simpler names the model expected (e.g., `age` instead of `adopter_age`, `housing` instead of `adopter_housing_type`).
  - Initially, the feature set provided to the model was reduced based on an error message. However, to improve recommendation score diversity (addressing an issue where all recommendations showed 100% match scores), the application was updated to provide the model with its full, intended feature set as defined in the global `MODEL_NUMERICAL_FEATURES` and `MODEL_CATEGORICAL_FEATURES` lists (using the corrected names).
- **Unique LLM Explanations:** Resolved an issue where the same LLM-generated match explanation was displayed for all recommended pets. This was fixed by:
  - Modifying the `@st.cache_data` decorated function `get_match_explanation` to include an additional parameter (`pet_id_for_cache_key`).
  - Passing a unique pet identifier (Animal ID) to this parameter for each pet, ensuring Streamlit's caching mechanism treats each call as distinct, thus generating a unique explanation per pet.

IV. Core Project Artifacts and Scripts

This section lists the key files, data, and scripts that constitute the project:

- **Input Data for Model Training:** `train_pairs.parquet/` (generated by `PairGCPLoad.py`, located at `gs://pairing-demo-bucket/train_pairs.parquet/`)
- **Adopter Data:** `gs://pairing-demo-bucket/Adopters/` (initially from `/Users/jeremy/Downloads/Adopter Dataset Creation.ipynb`)
- **Pet Data:** `gs://pairing-demo-bucket/pets_silver/` (original source, e.g., `/Users/jeremy/Downloads/drive-down`
- **Data Processing Script (Pandas):** `PairGCPLoad.py` (located in `/Users/jeremy/BigDataProj/Pairing/`)
- **Model Training Script:** `03_train_wd_model.py` (located in `/Users/jeremy/BigDataProj/model_training/`)
- **Trained Model File:** `pet_model.h5` (located at `gs://pairing-demo-bucket/models/pet_model.h5`)
- **Streamlit Application Script:** `app.py` (located in `/Users/jeremy/BigDataProj/streamlit_app/`)
- **Supporting Local Notebooks (Prototyping/Initial Data Handling):**
  - `/Users/jeremy/Downloads/Adopter Dataset Creation.ipynb`
  - `/Users/jeremy/Downloads/Dataset_Creation_and_Loading.ipynb`

V. Running and Deploying the Application

A. Running Locally

1. **Prerequisites:**
   - Python 3.8+.

- Access to Google Cloud Storage for downloading initial datasets if not already localized.

2. **Setup:**
   - Clone the repository.
   - Navigate to the `streamlit_app` directory: `cd /Users/jeremy/BigDataProj/streamlit_app/`
   - Create a Python virtual environment and activate it: `bash python3 -m venv venv source venv/bin/activate`
   - Install required packages: `bash pip install -r requirements.txt`
   - **Download Models and Data:**
     - **Llama Model:** Download your chosen Llama GGUF model (e.g., `llama-3-8b-instruct.Q4_K_M.gguf`). Place it in the `streamlit_app/models/` directory. If you place it elsewhere or use a different name, set the `LLAMA_MODEL_PATH` environment variable before running the app (e.g., `export LLAMA_MODEL_PATH=/path/to/your/model.gguf`).
     - **Recommendation Model:** Ensure `pet_model.h5` is in the `streamlit_app/` directory (or download from `gs://pairing-demo-bucket/models/pet_model.h5`).
     - **Pet and Adopter Data:** Ensure the `pets_silver_local` and `adopters_local` Parquet datasets are in the `streamlit_app/data/` directory (or download from `gs://pairing-demo-bucket/pets_silver/` and `gs://pairing-demo-bucket/Adopters/` and place them accordingly, i.e. `streamlit_app/data/pets_silver_local/pets_silver` and `streamlit_app/data/adopters_local/Adopters`).

3. **Run the Application:**
   - From the `streamlit_app` directory: `bash streamlit run app.py`

B. Deployment Considerations (General Notes)

- **Containerization:** Using Docker is a recommended approach to package the Streamlit application, its dependencies (including `llama-cpp-python`), and potentially the Llama model file.
  - The Dockerfile would need to handle the installation of `llama-cpp-python`. For GPU support in deployment, `llama-cpp-python` often needs to be compiled with specific flags (e.g., CUDA support) matching the deployment hardware.
- **Llama Model Management:**
  - The Llama GGUF model file is large. It can be included in the Docker image (increasing image size) or downloaded from a central storage (like GCS/S3) when the container starts.
  - The `LLAMA_MODEL_PATH` environment variable in `app.py` allows you to specify the model's location in the deployed environment.
- **Hardware:** Ensure the deployment environment has sufficient RAM (16GB+ is recommended for an 8B model) and CPU resources. For better performance, GPU acceleration is highly beneficial.
- **Dependencies:** All Python packages listed in `requirements.txt` must be installable in the deployment environment.

This README provides a snapshot of the Pet-Match Recommender project's design, components, and the journey of its development.