Project Description:

The project tackles the **CTR (Click-Through Rate) Prediction** problem aiming to predict whether a user will click on an ad. It addresses a **classification problem** where the target variable is is_click, indicating whether a user clicked on an advertisement. The dataset provided in the repository is taken from the Kaggle website (https://www.kaggle.com/datasets/arashnic/ctr-in-advertisement/data) and it includes features such as user demographics, session details, and product characteristics.

Key Steps in the Notebook:

1. **Data Preprocessing:**
    - The dataset is loaded and examined for missing values.
    - **Handling Missing Data:**
        - A column with excessive missing values (product_category_2) is dropped.
        - Missing values in other important columns ('user_group_id', 'gender', 'age_level', 'user_depth') are handled by removing rows with null values.
        - For city_development_index, missing values are imputed using random sampling of predefined categories.
    - Feature datetime is converted to a proper timestamp format.
2. **Feature Engineering:**
    - The hour of the day is extracted from the datetime column and added as a new feature.
    - The DictVectorizer is used to convert categorical features into a numeric format suitable for model training.
    - Features are standardized using StandardScaler to normalize numeric values.
3. **Feature Importance:**
    - The **Mutual Information (MI)** score is calculated for each feature to evaluate its importance in predicting is_click.
    - Features like session_id, user_id, and datetime have higher importance, while others contribute less to the prediction.
4. **Train/Test Split:**
    - The data is split into training and testing sets (80% training, 20% testing).
    - The target variable (is_click) is separated from the feature dataset for training the model.
5. **Model Training:**

- Several classifiers are used, including:
    - **Logistic Regression**
    - **Stochastic Gradient Descent (SGD)**
    - **Decision Tree**
    - **Random Forest**
    - **XGBoost**
- Each model is evaluated using **5-fold cross-validation** with the **F1 score** as the evaluation metric.

6. **Hyperparameter Tuning:**
    - For the **XGBoost classifier**, grid search is performed to optimize hyperparameters like n_estimators, learning_rate, max_depth, and min_child_weight.
    - The tuned model achieves a **Best F1 Score of ~0.1508** on the training set.
7. **Model Evaluation on Test Data:**
    - The optimized **XGBoost model** is trained on the entire training set.
    - It is then evaluated on the test set, achieving a **Test F1 Score of ~0.1542**.
8. **Model Serialization:**
    - The trained models and preprocessors (DictVectorizer, StandardScaler, and the XGBoost classifier) are saved as .pkl files for deployment or reuse.

Key Insights:

- **Data Imbalance**: The dataset suffers from class imbalance, as evidenced by the low F1 scores for most models. The XGBoost classifier addresses this by using the scale_pos_weight parameter.
- **Feature Selection**: Some features, like session_id and user_id, have higher predictive importance. Others, like gender and user_depth, contribute minimally.
- **Model Selection**: XGBoost outperformed other models, showing its robustness in handling complex data structures and imbalanced datasets.

Additional Scripts

This project includes two Python scripts to complement the Jupyter Notebook for streamlined usage and deployment.

1. ad-click-prediction.py

- **This script trains the best model (XGBoost) using the hyperparameters found during fine-tuning.**
- **The trained models and preprocessors are saved into .pkl files for reuse:**
    - **dict_vectorizer.pkl: Stores the vectorizer for categorical features.**
    - **standard_scaler.pkl: Stores the scaler for numerical features.**
    - **xgb_model.pkl: Stores the trained XGBoost model.**

2. test-predict.py

- **This script:**
    1. **Loads the saved .pkl files for preprocessing and the trained model.**
    2. **Takes an example row from the test dataset.**
    3. **Makes a prediction for the example row.**
    4. **Prints the prediction result (e.g., whether the ad will be clicked or not).**

These scripts streamline the workflow, allowing seamless training and testing without requiring the notebook.

Deployment:

Preprocessors and the model are saved in .pkl format for easy reuse.
This project includes a **Dockerized Service** and a Python client for making predictions.

Files:

1. **Dockerfile**: It creates a containerized environment to serve predictions using P**ython 3.12-slim** as the base image. All required libraries are installed via pipenv virtual environment. It sets /app as the working directory within the container and it copies the model files, service.py and virtual environment files to the working directory in the container. It exposes port 9696 for the web service, and for the entrypoint, it uses waitress-serve to host the Flask service defined in service.py.
2. **service.py**: Implements a web service to process requests.
3. **predict_client.py**: Sends prediction requests to the service.

How to Run:

1. Build the Docker image:

   docker build -t adpred .

2. Run the container:

docker run -d -p 9696:9696 adpred

3. Run the client to make a prediction:

   python predict_client.py

This setup ensures the model is deployable and ready to make predictions in a production-like environment.

Updated Virtual Environment Instructions

For better project reproducibility and ease of dependency management, the project uses **Pipenv**. If you want to run the project locally without Docker:

1. **Set up the Environment:** Install pipenv if not already installed:

   pip install pipenv

2. **Install Dependencies:** Use the Pipfile and Pipfile.lock to install dependencies:

   pipenv install --dev

3. **Activate the Virtual Environment:** Enter the virtual environment:

   pipenv shell

4. **Run the Service Locally:**

   python service.py

5. **Test the Prediction Locally:**

   python predict_client.py

Further Improvements

To enhance the model's performance, the following steps are recommended:

- **Sampling Techniques:** Use SMOTE or hybrid sampling to mitigate class imbalance.
- **Models:** Experiment with more models, voting classifier and deep learning methods.
- **Feature Engineering:** Try different attribute combinations, cyclic encoding and TimeSeriesSplit cross validation.
- **Deployment Readiness:** Implement monitoring and scalability optimizations for real-world deployment.