01-data-preprocessing

June 10, 2024

Import libraries

[ ]:
```
import os
import time
import pickle
import pandas as pd
import pyarrow.parquet as pa
from sklearn.feature_extraction import DictVectorizer
```

[ ]:
```
import warnings
warnings.filterwarnings('ignore')
```

Start time

[ ]:
```
start_time = time.time()
```

Setting path to the data directory

[ ]:
```
CURRENT_DIRECTORY = os.getcwd()
PARENT_DIRECTORY = os.path.dirname(CURRENT_DIRECTORY)
DATA_PATH = os.path.join(PARENT_DIRECTORY, '_data')
PICKLE_PATH = os.path.join(CURRENT_DIRECTORY, '_pickle')
DEST_PATH = os.path.join(CURRENT_DIRECTORY, 'DEST_PATH')  # Create a specific␣
↪directory
```

Ensure the DEST_PATH directory exists

[ ]:
```
if not os.path.exists(DEST_PATH):
    print(f"Creating directory: {DEST_PATH}")
    os.makedirs(DEST_PATH)
else:
    print(f"Directory already exists: {DEST_PATH}")
```

[ ]:
```
vectorise = DictVectorizer()
```

Notes: 1. We shall use the code of Data Pre-processing written for Week-01. 2. Here we are using Yellow taxi data of January, February, and March months. 3. train => January, validation => February, test => March.

[ ]:
```
def path_join(train, val, test):
    """
    Join the paths for the train, validation, and test datasets.
    Args:
    train (str): Filename for the train dataset.
    val (str): Filename for the validation dataset.
    test (str): Filename for the test dataset.
    Returns:
    list: List containing the full paths for the train, validation, and test␣
 ↪datasets.
    """
    train_data_path = os.path.join(DATA_PATH, train)
    val_data_path = os.path.join(DATA_PATH, val)
    test_data_path = os.path.join(DATA_PATH, test)
    return [train_data_path, val_data_path, test_data_path]
```

[ ]:
```
def read_data(data):
    """
    Read the data from a file and return it as a pandas DataFrame.
    Args:
    data (str): Path to the data file.
    Returns:
    pd.DataFrame: DataFrame containing the data.
    """
    if data.endswith('.parquet'):
        data = pa.read_table(data)
        df = data.to_pandas()  # Converting to pandas DataFrame
        df.columns = df.columns.str.lower()
        return df
    elif data.endswith('.csv'):
        df = pd.read_csv(data)
        df.columns = df.columns.str.lower()
        return df
    else:
        return 'Not valid format'
```

[ ]:
```
def save_pickle(obj, filename: str):
    """
    Save an object to a pickle file.
    Args:
    obj: Object to be saved.
    filename (str): Name of the file where the object will be saved.
    """
    with open(filename, "wb") as f_out:
        return pickle.dump(obj, f_out)
```

[ ]:
```python
def calculate_duration(data):
    """
    Calculate the duration of each trip in minutes.
    Args:
    data (pd.DataFrame): DataFrame containing the trip data.
    Returns:
    pd.DataFrame: DataFrame with an added 'duration' column.
    """
    data['duration'] = pd.to_datetime(data['lpep_dropoff_datetime']) - pd.
 ↪to_datetime(data['lpep_pickup_datetime'])
    data['duration'] = data['duration'].dt.total_seconds() / 60  # Convert␣
 ↪seconds to minutes
    return data
```

[ ]:
```python
def outliers(data):
    """
    Filter out trips with durations outside the range [1, 60] minutes.
    Args:
    data (pd.DataFrame): DataFrame containing the trip data.
    Returns:
    pd.DataFrame: DataFrame with outliers removed.
    """
    data_outliers = data[(data['duration'] >= 1) & (data['duration'] <= 60)]
    data_outliers['pulocationid'] = data_outliers['pulocationid'].astype(str)
    data_outliers['dolocationid'] = data_outliers['dolocationid'].astype(str)
    return data_outliers
```

[ ]:
```python
def convert_to_dict(data_outliers):
    """
    Convert the DataFrame to a list of dictionaries for vectorization.
    Args:
    data_outliers (pd.DataFrame): DataFrame containing the filtered data.
    Returns:
    list: List of dictionaries representing the data.
    """
    return data_outliers[['pulocationid', 'dolocationid', 'trip_distance']].
 ↪to_dict(orient='records')
```

[ ]:
```python
def fit_transform_(df_dict):
    """
    Fit and transform the data using DictVectorizer.
    Args:
    df_dict (list): List of dictionaries representing the data.
    Returns:
    scipy.sparse.csr_matrix: Transformed data.
    """
    return vectorise.fit_transform(df_dict)
```

[ ]:
```
def fit_(df_dict):
    """
    Transform the data using an already fitted DictVectorizer.
    Args:
    df_dict (list): List of dictionaries representing the data.
    Returns:
    scipy.sparse.csr_matrix: Transformed data.
    """
    return vectorise.transform(df_dict)
```

[ ]:
```
def pre_processing(data, choice):
    """
    Pre-process the data by calculating duration, removing outliers, and␣
 ↪vectorizing the data.
    Args:
    data (pd.DataFrame): DataFrame containing the trip data.
    choice (int): Choice for vectorization (0 for training data, 1 for␣
 ↪validation/test data).
    Returns:
    tuple: Tuple containing the vectorized data and the DataFrame with outliers␣
 ↪removed.
    """
    data = calculate_duration(data)
    data_outliers = outliers(data)
    df_dict = convert_to_dict(data_outliers)
    if choice == 0:
        X_train = fit_transform_(df_dict)
        return X_train, data_outliers
    elif choice == 1:
        X_val = fit_(df_dict)
        return X_val, data_outliers
    else:
        return 'Enter Choice 0 or 1'
```

[ ]:
```
def main(train, val, test):
    """
    Main function to execute the data pre-processing pipeline.
    Args:
    train (str): Filename for the train dataset.
    val (str): Filename for the validation dataset.
    test (str): Filename for the test dataset.
    """
    data_path_files = path_join(train, val, test)
    df_train = read_data(data_path_files[0])  # Read January data
    df_val = read_data(data_path_files[1])  # Read February data
    df_test = read_data(data_path_files[2])  # Read March data
    X_train, df_train = pre_processing(df_train, choice=0)
```

```
    X_val, df_val = pre_processing(df_val, choice=1)
    X_test, df_test = pre_processing(df_test, choice=1)
    y_train = df_train['duration']
    y_val = df_val['duration']
    y_test = df_test['duration']

    # Save DictVectorizer and datasets
    save_pickle(vectorise, os.path.join(DEST_PATH, "vectorise.pkl"))
    save_pickle((X_train, y_train), os.path.join(DEST_PATH, "train.pkl"))
    save_pickle((X_val, y_val), os.path.join(DEST_PATH, "val.pkl"))
    save_pickle((X_test, y_test), os.path.join(DEST_PATH, "test.pkl"))
```

[ ]:
```
if __name__ == '__main__':
    # File Names
    january_file_name = 'green_tripdata_2023-01.parquet'
    february_file_name = 'green_tripdata_2023-02.parquet'
    march_file_name = 'green_tripdata_2023-03.parquet'
    main(january_file_name, february_file_name, march_file_name)

    # End time
    end_time = time.time()
    print(f"Total time taken to run the script: {end_time - start_time}␣
 ↪seconds")
```

02-train

June 10, 2024

[ ]:
```
import os, pickle, mlflow, logging
from sklearn.ensemble import RandomForestRegressor
from sklearn.metrics import mean_squared_error
```

Configure logging

[ ]:
```
logging.basicConfig(filename='logs/training.log', level=logging.INFO)
```

Command to run for mlflow mlflow ui –backend-store-uri sqlite:///mlflow.db

Define functions

[ ]:
```
def load_pickle(fileName: str):
    """
    Load data from a pickle file.
    Args:
    fileName (str): Path to the pickle file.
    Returns:
    object: Data loaded from the pickle file.
    """
    try:
        with open(fileName, 'rb') as f:
            return pickle.load(f)
    except FileNotFoundError:
        logging.error(f"Error: File '{fileName}' not found.")
        return None
```

[ ]:
```
def train_(Data_path: str = 'DEST_PATH', max_depth: int = 10, random_state: int␣
 ↪= 0):
    """
    Train a random forest regressor model.
    Args:
    Data_path (str): Path to the directory containing data files.
    max_depth (int): Maximum depth of the trees in the random forest.
    random_state (int): Seed used by the random number generator.
    """
    mlflow.sklearn.autolog()

    # Load training and validation data
```

```
    X_train, y_train = load_pickle(os.path.join(Data_path, 'train.pkl'))
    X_val, y_val = load_pickle(os.path.join(Data_path, 'val.pkl'))

    # Convert target variables to numpy arrays
    y_train = y_train.to_numpy()
    y_val = y_val.to_numpy()

    # Start MLflow run
    with mlflow.start_run():
        logging.info("Training random forest regressor model...")
        # Initialize and train random forest regressor model
        rf = RandomForestRegressor(max_depth=max_depth,␣
↪random_state=random_state)
        rf.fit(X_train, y_train)
        y_pred = rf.predict(X_val)

        # Calculate root mean square error
        rmse = mean_squared_error(y_val, y_pred, squared=False)
        logging.info(f'Root Mean Square Error = {rmse}')
```

Entry point of the script

[ ]:
```
if __name__ == '__main__':
    # Set the path to the data directory
    CURRENT_DIRECTORY = os.getcwd()
    DEST_PATH = os.path.join(CURRENT_DIRECTORY, 'DEST_PATH')

    # Train the model using the data in DEST_PATH
    train_(DEST_PATH)
```

03-hypo

June 10, 2024

[ ]:
```
import os, pickle, mlflow, logging
import numpy as np
```

[ ]:
```
from hyperopt import STATUS_OK, Trials, fmin, hp, tpe
from hyperopt.pyll import scope
```

[ ]:
```
from sklearn.ensemble import RandomForestRegressor
from sklearn.metrics import mean_squared_error
```

Configure logging

[ ]:
```
logging.basicConfig(filename='logs/hypo.log', level=logging.INFO)
```

[ ]:
```
mlflow.set_tracking_uri("http://127.0.0.1:5000")
mlflow.set_experiment("random-forest-hyperopt")
```

Define functions

[ ]:
```
def load_pickle(fileName: str):
    """
    Load data from a pickle file.
    Args:
    fileName (str): Path to the pickle file.
    Returns:
    object: Data loaded from the pickle file.
    """
    try:
        with open(fileName, 'rb') as f:
            return pickle.load(f)
    except FileNotFoundError:
        logging.error(f"Error: File '{fileName}' not found.")
        return None
```

[ ]:
```
def optimisation_(Data_path: str = 'DEST_PATH', num_trails = int):

    # Load training and validation data
    X_train, y_train = load_pickle(os.path.join(Data_path, 'train.pkl'))
    X_val, y_val = load_pickle(os.path.join(Data_path, 'val.pkl'))
```

```
    # Convert target variables to numpy arrays
    y_train = y_train.to_numpy()
    y_val = y_val.to_numpy()
    def objective(params):
        # Start MLflow run
        with mlflow.start_run():
            logging.info("Training random forest regressor model...")
            mlflow.log_params(params)
            # Initialize and train random forest regressor model
            rf = RandomForestRegressor(**params)
            rf.fit(X_train, y_train)
            y_pred = rf.predict(X_val)

            # Calculate root mean square error
            rmse = mean_squared_error(y_val, y_pred, squared=False)
            mlflow.log_metric("rmse", rmse)
            logging.info(f'Root Mean Square Error = {rmse}')
        return {'loss':rmse, 'status':STATUS_OK}
    search_space = {
        'max_depth' : scope.int(hp.quniform('max_dept', 1,20,1)),
        'n_estimators': scope.int(hp.quniform('n_estimator', 10,50,1)),
        'min_samples_split': scope.int(hp.quniform('min_samples_split',␣
↪2,10,1)),
        'random_state':42
    }
    rstate = np.random.default_rng(42)  # For Reproducable Results
    fmin(
        fn=objective,
        space=search_space,
        algo=tpe.suggest,
        max_evals=num_trails,
        trials=Trials(),
        rstate=rstate
    )
```

Entry point of the script

[ ]:
```
if __name__ == '__main__':
    # Set the path to the data directory
    CURRENT_DIRECTORY = os.getcwd()
    DEST_PATH = os.path.join(CURRENT_DIRECTORY, 'DEST_PATH')

    # Train the model using the data in DEST_PATH
    optimisation_(DEST_PATH, num_trails=30)
```

04-register-model

June 10, 2024

[ ]:
```
import os
import pickle
import logging
import mlflow
import click
from mlflow.entities import ViewType
from mlflow.tracking import MlflowClient
from sklearn.ensemble import RandomForestRegressor
from sklearn.metrics import mean_squared_error
```

Configure logging

[ ]:
```
logging.basicConfig(filename='logs/model_register.log', level=logging.INFO)
```

[ ]:
```
HPO_EXPERIMENT_NAME = "random-forest-hyperopt"
EXPERIMENT_NAME = "random-forest-best-models"
RF_PARAMS = ['max_depth', 'n_estimators', 'min_samples_split',␣
 ↪'min_samples_leaf', 'random_state']
```

Set MLflow tracking URI and experiment name

[ ]:
```
mlflow.set_tracking_uri("http://127.0.0.1:5000")
mlflow.set_experiment(EXPERIMENT_NAME)
mlflow.sklearn.autolog()
```

[ ]:
```
def load_pickle(fileName: str):
    """
    Load data from a pickle file.
    Args:
    fileName (str): Path to the pickle file.
    Returns:
    object: Data loaded from the pickle file.
    """
    try:
        with open(fileName, 'rb') as f:
            return pickle.load(f)
    except FileNotFoundError:
        logging.error(f"Error: File '{fileName}' not found.")
```

```
        return None
```

[ ]:
```
def train_and_log_model(data_path, params):
    # Load data
    X_train, y_train = load_pickle(os.path.join(data_path, "train.pkl"))
    X_val, y_val = load_pickle(os.path.join(data_path, "val.pkl"))
    X_test, y_test = load_pickle(os.path.join(data_path, "test.pkl"))
    with mlflow.start_run():
        # Convert relevant parameters to int
        for param in RF_PARAMS:
            if param in params:
                params[param] = int(params[param])

        # Initialize and train RandomForestRegressor
        rf = RandomForestRegressor(**params)
        rf.fit(X_train, y_train)

        # Evaluate model on validation and test sets
        val_rmse = mean_squared_error(y_val, rf.predict(X_val), squared=False)
        mlflow.log_metric("val_rmse", val_rmse)
        test_rmse = mean_squared_error(y_test, rf.predict(X_test),␣
 ↪squared=False)
        mlflow.log_metric("test_rmse", test_rmse)
```

[1]:
```
def run_register_model(data_path: str, top_n: int):
    client = MlflowClient()

    # Retrieve the top_n model runs and log the models
    experiment = client.get_experiment_by_name(HPO_EXPERIMENT_NAME)
    if experiment is None:
        logging.error(f"Error: Experiment '{HPO_EXPERIMENT_NAME}' not found.")
        return None
    runs = client.search_runs(
        experiment_ids=experiment.experiment_id,
        run_view_type=ViewType.ACTIVE_ONLY,
        max_results=top_n,
        order_by=["metrics.rmse ASC"]
    )
    for run in runs:
        train_and_log_model(data_path=data_path, params=run.data.params)

    # Select the model with the lowest test RMSE
    experiment = client.get_experiment_by_name(EXPERIMENT_NAME)
    best_run = client.search_runs(
        experiment_ids=experiment.experiment_id,
        run_view_type=ViewType.ACTIVE_ONLY,
        max_results=top_n,
```

```
        order_by=["metrics.test_rmse ASC"]
    )[0]

    # Register the best model
    run_id = best_run.info.run_id
    model_uri = f"runs:/{run_id}/model"
    mlflow.register_model(model_uri, name="rf-best-model")
```

[ ]:
```
if __name__ == '__main__':
    # Set the path to the data directory
    CURRENT_DIRECTORY = os.getcwd()
    DEST_PATH = os.path.join(CURRENT_DIRECTORY, 'DEST_PATH')

    run_register_model(DEST_PATH, top_n=5)
```