**Steps Involved in Resource Creation :**

* **Setup Project: Set up a Conda virtual environment and install dependencies listed in requirements.txt.**
* **Project Structure: Build project folder structure using template.py and organize pipeline stages.**
* **Configuration Files: Configure dependencies in requirements.txt and package the project with pyproject.toml.**
* **Logging & Error Handling: Implement custom logging in logger.py and error handling in exception.py.**
* **Constants: Define commonly used constants in constants.py.**
* **Entity Files: Create config\_entity.py for pipeline configurations and artifact\_entity.py for stage outputs.**
* **Database & Data Handling: Use mongo\_db\_connection.py for MongoDB connection, data\_dump.py to upload data, and usvisa\_data.py for data export.**
* **Database & Data Handling (Mongodb):**
* **mongo\_db\_connection.py: Establish connection to MongoDB using pymongo.**
* **data\_dump.py: Upload manually downloaded CSV data into MongoDB.**
* **data\_access.usvisa\_data.py: Access MongoDB data and export it back to CSV for further processing.**
* **created various pipeline components with the help of all the above resources created.**
* **Data Ingestion: Fetch and split data into train/test, save to file, and generate DataIngestionArtifact.**
* **Data Transformation: Apply transformations and SMOTE for balancing, save transformed data and preprocessor, return DataTransformationArtifact.**
* **Data Validation: Validate data schema and detect drift, return validation results in DataValidationArtifact.**
* **Model Trainer: Train model, generate metrics (accuracy, F1), and save the model, returning ModelTrainerArtifact.**
* **Model Evaluation: Compare trained model against production model, decide whether to accept the new model, return ModelEvaluationArtifact.**
* **Model Pusher: Push accepted model to S3, generate ModelPusherArtifact confirming successful upload.**
* **Key Files to Update/Modify for Each Component you create:**
* **config\_entity.py: Defines configurations for each pipeline stage.**
* **artifact\_entity.py: Defines the outputs (artifacts) of each pipeline stage.**
* **constants.py: Stores commonly used constants.**
* **.env: Loads environment-specific variables (e.g., database credentials, paths).**
* **requirements.txt: Lists dependencies for the project.**
* **pyproject.toml: Specifies metadata for packaging the project.**
* **logger.py: Custom logging setup.**
* **exception.py: Custom error handling and exception management.**
* **Steps of various pipeline components in detail:**

**1. data\_ingestion.py**

* **export\_data\_into\_feature\_store**
  + Fetches data from MongoDB.
  + Saves the data into a CSV file in the feature store directory.
* **split\_data\_as\_train\_test**
  + Splits the data into training and testing sets using train\_test\_split.
  + Exports training and testing sets to respective file paths.
* **initiate\_data\_ingestion**
  + Combines the above steps.
  + Returns DataIngestionArtifact with paths to training and testing datasets.

**2. data\_transformation.py**

* **get\_data\_transformer\_object**
  + Creates a ColumnTransformer pipeline for:
    - **One-Hot Encoding** for categorical variables.
    - **Ordinal Encoding** for ordinal variables.
    - **Power Transformation** for skewed variables.
    - **Standard Scaling** for numerical variables.
* **initiate\_data\_transformation**
  + Reads training and testing datasets.
  + Computes additional features like company\_age.
  + Drops specified columns from the dataset.
  + Transforms the dataset using the preprocessor object.
  + Balances the dataset using SMOTEENN.
  + Saves the transformed data and preprocessor object.
  + Returns DataTransformationArtifact.

**3. data\_validation.py**

* **validate\_number\_of\_columns**
  + Validates if the number of columns in the dataset matches the schema.
* **is\_column\_exist**
  + Checks the presence of required numerical and categorical columns.
* **detect\_dataset\_drift**
  + Compares the training and testing datasets for drift using evidently.
  + Saves the drift report in YAML format.
* **initiate\_data\_validation**
  + Executes the above validation checks.
  + Detects data drift between training and testing datasets.
  + Returns DataValidationArtifact with validation status and drift report.

**4. model\_trainer.py:**

* **Step**: **Model Training**  
  **Functions Used**:
  + get\_model\_object\_and\_report(): Trains a model using neuro\_mf to get the best model and its metrics (accuracy, F1 score, precision, recall).
  + initiate\_model\_trainer(): Loads the transformed data, trains the best model, and saves the trained model and its metrics.
* **Key Process**:
  + Loads transformed train and test data.
  + Uses ModelFactory to get the best model and its performance metrics.
  + If the model meets the expected accuracy, it creates and saves the model object with preprocessor and model details.

**5. model\_evaluation.py:**

* **Step**: **Model Evaluation**  
  **Functions Used**:
  + get\_best\_model(): Checks if the production model is available in the S3 bucket and loads it if present.
  + evaluate\_model(): Compares the F1 score of the newly trained model with the best production model. Returns whether the new model is accepted based on performance.
  + initiate\_model\_evaluation(): Runs the model evaluation, returns the evaluation result, and prepares the ModelEvaluationArtifact.
* **Key Process**:
  + Loads test data and computes F1 scores for both trained and best production models.
  + Compares performance and decides if the new model is accepted.
  + Creates and returns a ModelEvaluationArtifact based on evaluation results.

**6. model\_pusher.py:**

* **Step**: **Model Pushing**  
  **Functions Used**:
  + initiate\_model\_pusher(): If the model is accepted, uploads the model artifacts to the S3 bucket.
* **Key Process**:
  + Uses USvisaEstimator to upload the trained model to the specified S3 location.
  + Creates and returns a ModelPusherArtifact after successfully uploading the model.