**Database Systems Project Part IV**

**End-to-End Solution Integration and Data-Driven / Database Programming**

Ming Gao (mg7572) Wes Wang (lw2534) Zejun Zhang (zz4140)

# **1. Business Use Cases**

An insurance company that has provided Health Insurance to its customers now needs some help in building a model to predict whether the customers from past year will also be interested in Vehicle Insurance provided by the company. Building a model to predict whether a customer would be interested in Vehicle Insurance is extremely helpful for the company because it can then accordingly plan its communication strategy to reach out to those customers and optimize its business model and revenue. Therefore, the company aims to make precise predictions via machine learning models to target potential customers who may be interested, in order to promote their products more efficiently and save costs.

# **2. Data-driven End-to-End Development**

## **2.1. Introduction**

In our project, we've developed a data-driven web application that seamlessly integrates data collection and machine learning. Our web application mainly enables data addition and machine learning prediction based on user commands. Typically, it enables seamless re-training of machine-learning models upon changes to the corresponding source of unstructured input data. To enhance user experience, the application features voice input capabilities. This input is then processed through speech recognition on the backend, transforming spoken words into the structured format needed for our purposes. Currently, our focus is on simulating post-speech recognition scenarios.

This document provides a comprehensive guide for the design, implementation, optimization, and management of a data-driven end-to-end application, aligning IT strategy with business objectives. Here, we would like to show the high-level structure of our Reference Architecture document as follows:

* Business Layer in Organizing Framework — Section 1
* RA Foundational Principles — Section 2.2
* Data Layer in Organizing Framework — Section 2.3
* Application Layer and Technology Layer in Organizing Framework — Section 2.4 & 2.5
* Data Governance — Section 2.6
* Implementation Examples — Section 3
* Appendix (Github link, Database Schemas, ORM Implementation Code Snippets)

## **2.2. Foundational Principles**

**Integration of Data Collection and Machine Learning.** The architecture emphasizes seamless integration between data collection and machine learning processing, facilitating efficient data flow and processing.

**Dynamic Data Management and Storage.** Emphasizes robust data management with MySQL databases, structured for efficient storage, retrieval, and processing of both training and test datasets.

**Efficient Data Processing and Structuring.** It incorporates mechanisms for transforming unstructured sentence inputs into structured data points necessary for machine learning, ensuring the accuracy and utility of the data, crucial for the effective functioning of machine learning models.

**Query Optimization and Performance Monitoring.** Prioritizes optimization of database queries and continuous performance monitoring to maintain high efficiency and responsiveness of the system.

**Frontend.** Facilitates elegant and efficient request logic, simple and straightforward front-end UI, clear display of model performance results.

**Frontend-Backend Synchronization.** Ensures a cohesive connection between the frontend (user interface) and backend (server-side operations), facilitated by frameworks like Flask, for real-time data processing and response.

**Predictive Analytics Using Advanced Machine Learning Techniques.** Utilizes state-of-the-art machine learning models, like XGBoost, for predictive analytics, enhancing the predictive accuracy of customer behavior.

**Object Relational Mapping.** Provides high-level abstractions of object and relational database entity mappings. Ensures developers to focus on business logic, requests, and semantics rather than complex and tedious SQL queries. Additionally, it also assumes and enforces consistency on both ends, database tables and app backend objects.

**Flexible Backend Operations.** The backend is capable of handling diverse functions, including data addition and predictive analysis, enhancing the application's versatility.

**Data Privacy and Security Compliance.** Maintains stringent adherence to data privacy regulations and employs encryption and access control to safeguard personal and sensitive data.

**Comprehensive Data Lifecycle Management.** Implements a holistic approach to managing the entire data lifecycle, from acquisition to disposal, ensuring data relevance and minimizing redundancy.

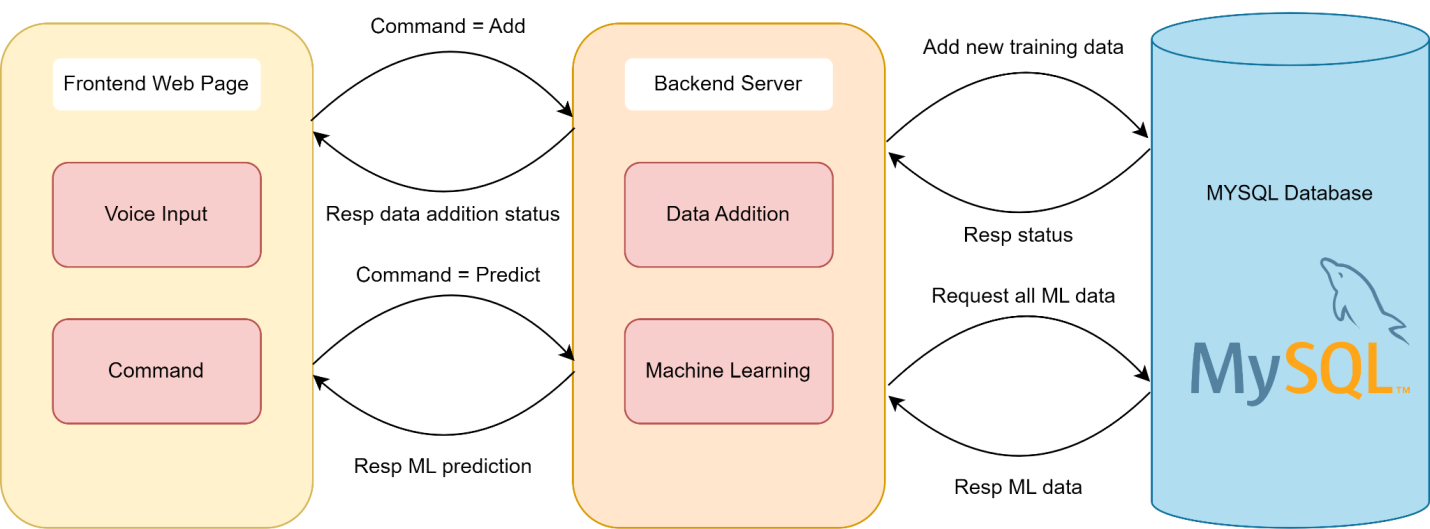
**Scalability and Flexibility.** The architecture is designed to be scalable to handle increasing data volumes and flexible to integrate future advancements in data processing and machine learning techniques.

## **2.3. Data Processing and Storage**

Our application specifically aims to predict Health Insurance owners’ interest in Vehicle Insurance. To achieve this, we collect users’ inputs, which are in the form of entire spoken sentences. Then we split the sentences into a variety of data points: ID, gender, age, driving license status, region code, previous insurance history, vehicle age, vehicle damage history, annual premium, policy sales channel, and length of time with the insurer (vintage). We also split the processed data into training data and test data for machine learning. Additionally, our training dataset includes a “Response” label, derived from the historical choices of previous customers, to inform our machine learning models as the target variable.

In terms of data storage, we deployed a MySQL database to manage our data. In particular, for machine learning, we created two tables (training\_data and test\_data) to store the training set and the test set respectively.

## **2.4. Overall Workflow**



**Figure.** Flow Chart of Our Full-stack Machine Learning Web Application

**Frontend Development.** We have two major functionalities in our application, adding data to our relational database and having a ML model predicting customer interest using the data retrieved from the database. Therefore, in our front-end page, we have two input areas corresponding to each of the two functionalities. We set appropriate styling using CSS to accommodate the length of the input.

When the users click on the “Retrieve” button, the front-end JavaScript logic will send a request to the backend with information entered by the users. Since our back-end server is deployed on the same machine as the front-end, we direct the request to a localhost endpoint. When the asynchronous requests receive their corresponding output, we will parse and display the JSON data into multiple lines down below the “Retrieve” button.

**Frontend-Backend Connection.** To connect our frontend and backend, we finally leveraged Flask, an efficient and lightweight framework based on Python. We implemented our functions based on this framework to ingest the parameters from the frontend and execute data addition or machine learning prediction in the backend. In detail, we applied multiple annotations supported by Flask, such as @app.route(methods=[‘POST’]) annotation to automatically track the POST requests from the frontend and parse the obtained JSON-format data into the parameters compatible with the inputs of the functions we implemented in the backend.

**Backend Development**. Our backend can achieve two functions: data addition and machine learning prediction respectively. After extracting the voice input and the user command from the front-end request, for convenience, we first divided the entire spoken sentence into the format compatible with machine learning datasets. Based on our practical experience as Data Science students, we leveraged the XGBoost model, one of the most advanced machine learning models, to predict the Health Insurance owners who will be interested in Vehicle Insurance.

If the user command is “Add”, the new data would be added after the last row of the training\_data table as a new record in the training set. Furthermore, the backend would send a message, ‘Data added to Training Set’, to the frontend.

If the user command is “Predict”, the training set and the test set would be retrieved from the database and then used for machine learning, which would send the performance of the XGBoost model (i.e. accuracy and AUC score on the test set) and the prediction of the interest status back to the frontend.

The detailed ML workflow is as follows: First, we would apply the One-Hot encoding to process the categorical features. Second, the SMOTE algorithm, a kind of over-sampling method, would be utilized to balance the imbalanced data since the ratio of the two original classes was approximately 7 to 1 for disinterest to interest. Third, we would train the XGBoost model on the over-sampled training dataset and calculate the accuracy and AUC score on the test set to obtain the model performance explicitly. Lastly, the trained model would be applied to make the prediction for the new data.

## **2.5. Object Relational Mapping (ORM)**

**SQLAlchemy.** Our goal is to implement an ORM functionality which can facilitate high-level abstraction of our mapping from python entity objects to entities in the relational database, and also vice versa. When integrating with Python Flask web application backend, SQLAlchemy is a good choice as it is lightweight, requires minimum programming and also has all the basic CRUD (create, read, update, delete) operations supported. After successful implementation of SQLAlchemy ORM, it will be easy for programmers to interact with the relational database, as they only need to worry about the high-level abstract business logic, but not the actual implementation detail when it comes to querying the database.

We first created a Python class for the purpose of training data abstraction, which we named TrainingData. Each attribute in the class represents a column in the database table for the training data. For complete database table schemas, please refer to the Appendix.

Then, when we need to add the data to the database, i.e. when a new piece of training data comes in, we first instantiate an instance of the class with the information provided by the incoming request. After we create such an instance, we call the SQLAlchemy out-of-the-box functions to save the information into the database. Please refer to the Appendix for the corresponding code. When we need to retrieve data from the database, it is the reverse process of what’s been described. The only difference is now that we also need a separate test data set, we need to create a separate class to host the instances of that testing data set. Then, we simply call the .query.all() function that comes with SQLAlchemy module, it returns all the data for our ML model.

**MyBatis Plus & SpringBoot.** To demonstrate our understanding and skills in ORM and also for the purpose of improving our full-stack project, we also explored other methods to implement ORM. Here, we explain our work using MyBatis Plus together with SpringBoot, which is a very common combination as a replacement for Hibernate + SpringBoot. We used it on our employee database, and it allows administrators from within the insurance company to manage employees. Please refer to the Appendix for the actual code of our implementation.

MyBatis Plus is not a usual ORM like Hibernate, as it is more of a persistence layer framework, but it has all the convenient features supported by any other ORMs, and it also comes with additional handy features like pagination, etc. We selected it because it is easy to incorporate, as it requires light programming after importing the module, and it works well with the basic CRUD operations as well. It is contained in the [EmployeeController.java](https://github.com/MingGao077/2433-Database-Systems/blob/main/Project-Part4/EmployeeController.java). Note that it does not have a separate file as it is closely integrated with Spring Boot controllers.

Each time we need to query the database, we simply wrap the Employee class instance with a LambdaQueryWrapper class and send the query request using the current session. It is fast and simple. For more advanced features, we also created an Employee service layer, which can host abstraction of the functionalities of the operations we want to perform on the Employee class when combining a service layer implementation. Then, suppose we want to add an employee to the database, we call the .save() function on the employee service object.

## **2.6. Data Governance and Query Optimization**

**Data Quality and Integrity**. Ensure the accuracy and completeness of user data collected, such as ID, gender, age, etc. This is vital for training effective machine learning models, as the accuracy of these models is directly dependent on the quality of input data.

**Data Privacy and Security**. Given the involvement of personal insurance information, it's quite imperative to comply with data privacy regulations (such as GDPR) and implement encryption and access control on the database side (MySQL) measures to protect the data from unauthorized access.

**Standardization and Data Integration**. Data inputs, like the conversion of voice input to structured data, were standardized to ensure consistency. This aids the machine learning models in processing and learning from the data more efficiently.

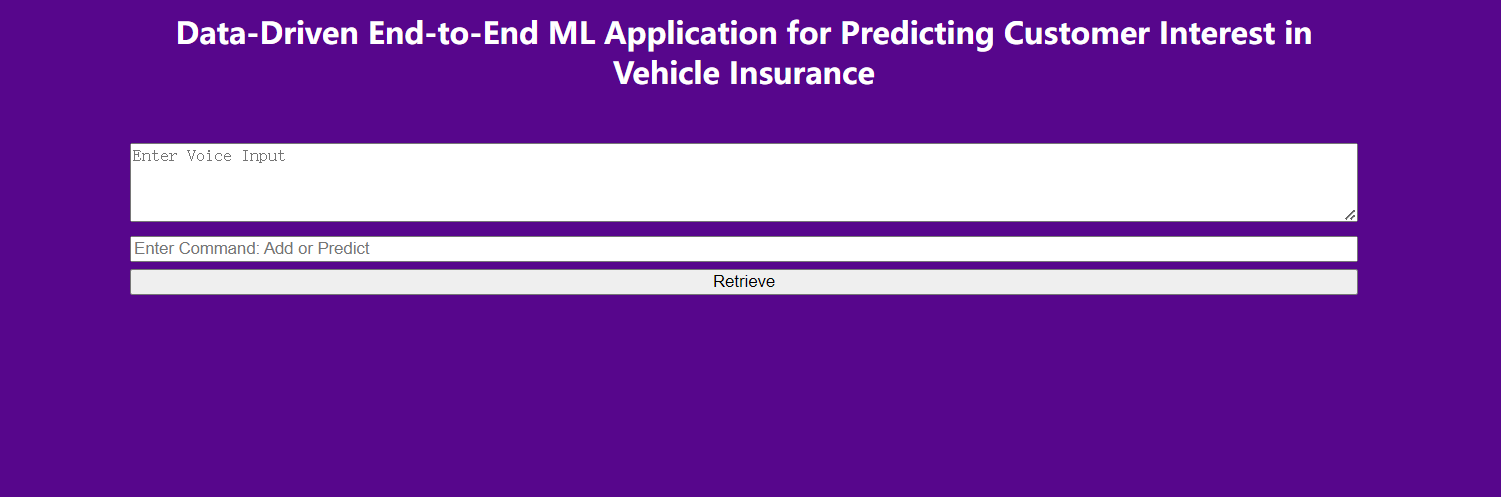
**Data Lifecycle Management.** Manage the entire lifecycle of data, from its creation and storage to its use and eventual disposal. This includes regularly reviewing and purging data that is no longer needed.

**Query Optimization and Performance Monitoring**. Since we deploy a MySQL database, as the number of customers increases, the database would be too large to search fast. Then, it is necessary to optimize database queries to enhance application performance and monitor the data processing process to identify and resolve issues promptly. The method we adopt here is to index the specific columns of the database, which can speed up the searching process by 100 times.

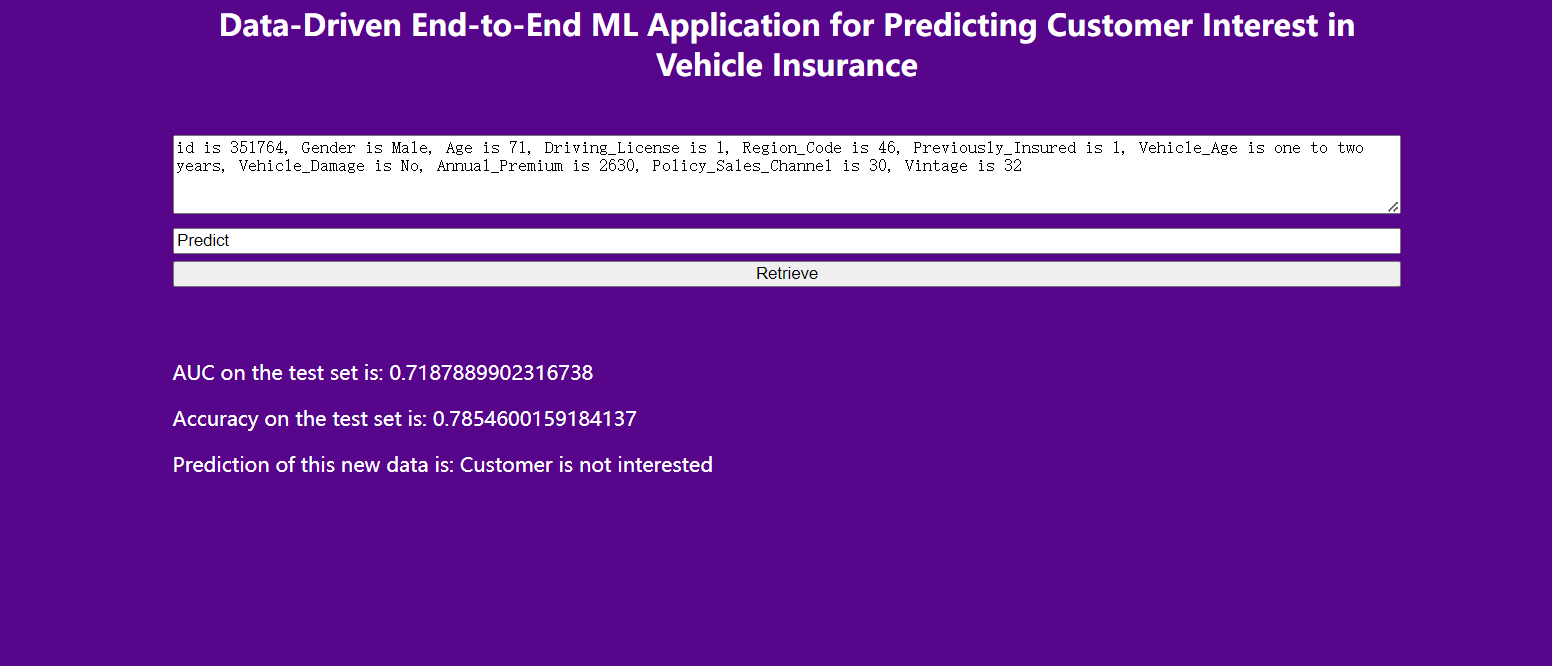
# **3. Implementation Examples**

Here, we would like to show some specific implementation examples to operate our web application.

When opening our web application, you can see the web page shown below:

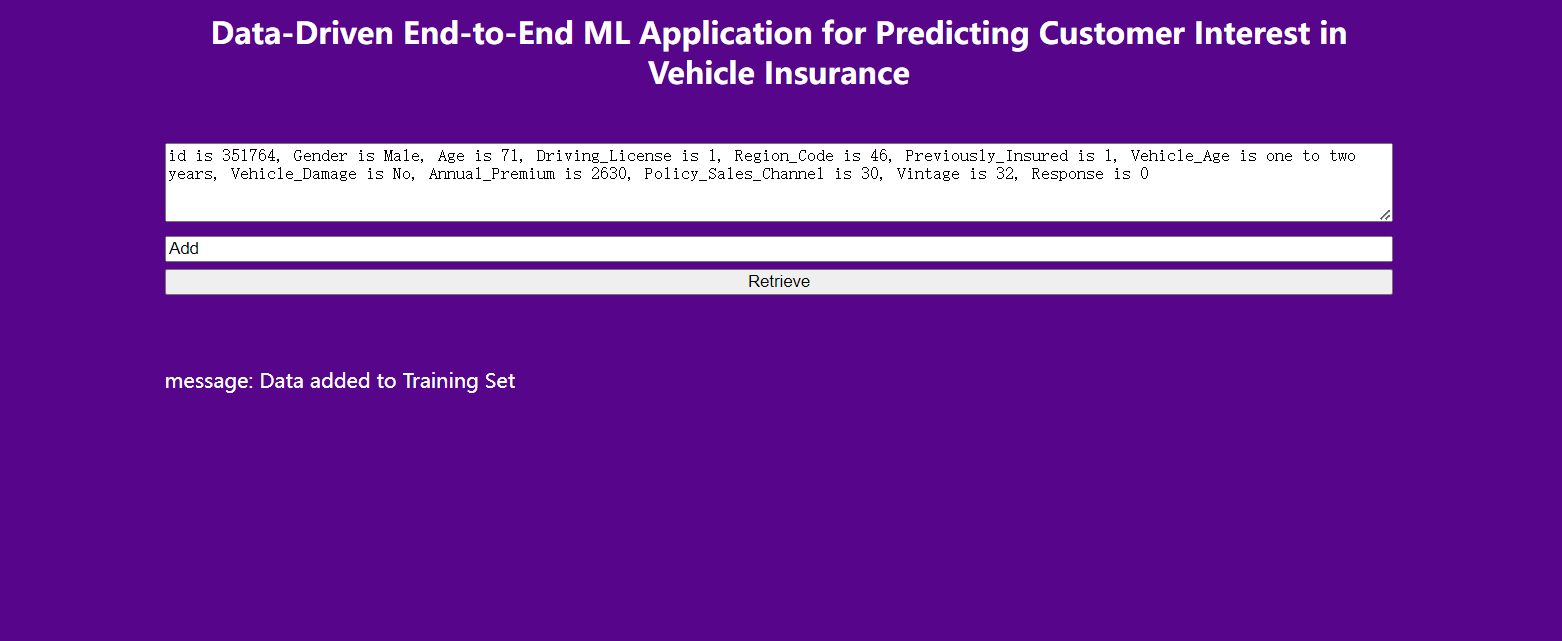


After typing the new input and the “Predict” command, click the “Retrieve” button, and you will get the following web page:



The web page would show the performance of the XGBoost model (i.e. AUC score and accuracy on the test set) and the prediction of the new input data. In this example, we can see that the XGBoost model achieves about 0.719 AUC and 0.785 accuracy on the test set, and it predicts that this health insurance owner (id = 351764) is not interested in the insurance company’s vehicle insurance.

After typing the new input and the “Add” command, click the “Retrieve” button, and you will get the following web page. Here, we would like to highlight that the new input here must include “Response” as the new input data would be converted to the format compatible with the ML training set.



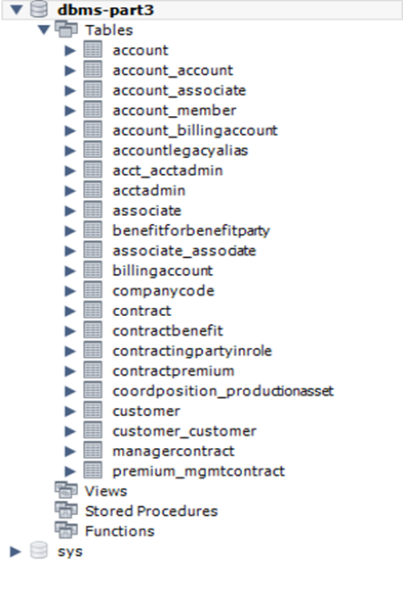
The website would show a message that the new input data is added to the ML training set (i.e. Table training\_data in MySQL database). Then, the database is well updated for the subsequent machine learning or data addition operations.

# **Appendix**

This work is stored in the main branch of the Github repository:

<https://github.com/MingGao077/2433-Database-Systems>

Since Zejun Zhang (zz4140) did the first three parts of this project solo, his previous work is stored in a branch (zejzhang) of this repository.

**MySQL Database Schema of Project-Part 3**

**SQLAlchemy ORM For Adding Data To DB**

|  |
| --- |
| **# ORM for adding data to the database  from flask import Flask from flask\_sqlalchemy import SQLAlchemy from flask\_cors import CORS  app = Flask(\_\_name\_\_) CORS(app)  # Configure SQLAlchemy app.config['SQLALCHEMY\_DATABASE\_URI'] = 'mysql+pymysql://root:1194@localhost/database\_project\_final' app.config['SQLALCHEMY\_TRACK\_MODIFICATIONS'] = False  db = SQLAlchemy(app)  class TrainingData(db.Model):  \_\_tablename\_\_ = 'training\_data'   id = db.Column(db.Integer, primary\_key=True)  Gender = db.Column(db.String(255))  Age = db.Column(db.Integer)  Driving\_License = db.Column(db.Integer)  Region\_Code = db.Column(db.Integer)  Previously\_Insured = db.Column(db.Integer)  Vehicle\_Age = db.Column(db.String(255))  Vehicle\_Damage = db.Column(db.String(255))  Annual\_Premium = db.Column(db.Integer)  Policy\_Sales\_Channel = db.Column(db.Integer)  Vintage = db.Column(db.Integer)  Response = db.Column(db.Integer) db.create\_all()  def add\_data\_to\_training\_orm(s):  data = parse\_new\_input(s)  training\_data = TrainingData(  id=int(data['id']),  Gender=data['Gender'].values[0],  Age=int(data['Age']),  Driving\_License=int(data['Driving\_License']),  Region\_Code=int(data['Region\_Code']),  Previously\_Insured=int(data['Previously\_Insured']),  Vehicle\_Age=data['Vehicle\_Age'].values[0],  Vehicle\_Damage=data['Vehicle\_Damage'].values[0],  Annual\_Premium=int(data['Annual\_Premium']),  Policy\_Sales\_Channel=int(data['Policy\_Sales\_Channel']),  Vintage=int(data['Vintage']),  Response=int(data['Response'])  )  db.session.add(training\_data)  db.session.commit()** |

**SQLAlchemy ORM For Retrieving Training Data from DB**

|  |
| --- |
| **# ORM for retrieving information from the database. from sqlalchemy import Column, Integer, String, Boolean class TestData(db.Model):  \_\_tablename\_\_ = 'test\_data'   db = SQLAlchemy(app)  id = Column(Integer, primary\_key=True)  Gender = Column(String(255))  Age = Column(Integer)  Driving\_License = Column(Integer)  Region\_Code = Column(Integer)  Previously\_Insured = Column(Integer)  Vehicle\_Age = Column(String(255))  Vehicle\_Damage = Column(String(255))  Annual\_Premium = Column(Integer)  Policy\_Sales\_Channel = Column(Integer)  Vintage = Column(Integer)  Response = Column(Integer)  # Create tables db.create\_all()  def perform\_machine\_learning\_orm():  # Retrieve data using SQLAlchemy  training\_data = TrainingData.query.all()  test\_data = TestData.query.all()   # Convert SQLAlchemy query results to pandas DataFrame  training\_data = pd.DataFrame([vars(entry) for entry in training\_data])  test\_data = pd.DataFrame([vars(entry) for entry in test\_data])   # Split data  X\_train = training\_data.drop(columns=['id', 'Response'], axis=1)  X\_test = test\_data.drop(columns=['id', 'Response'], axis=1)  y\_train = training\_data['Response']  y\_test = test\_data['Response']    # Get dummies of categorical features  train\_discrete\_features = X\_train.select\_dtypes('object').columns.to\_list()  for feature in train\_discrete\_features:  X\_train = encode\_and\_bind(X\_train, feature)  test\_discrete\_features = X\_test.select\_dtypes('object').columns.to\_list()  for feature in test\_discrete\_features:  X\_test = encode\_and\_bind(X\_test, feature)   # Process imbalanced data  over = SMOTE(sampling\_strategy='auto', random\_state=3)  X\_train\_o, y\_train\_o = over.fit\_resample(X\_train, y\_train)    # Machine Learning  xgb = XGBClassifier().fit(X\_train\_o,y\_train\_o)  xgb\_pred = xgb.predict(X\_test)    # Calculate evaluation metrics on the test set  XGBoost\_accuracy = accuracy\_score(y\_test, xgb\_pred)  XGBoost\_auc = roc\_auc\_score(y\_test, xgb\_pred)    return xgb, XGBoost\_accuracy, XGBoost\_auc, X\_train.columns** |

**MyBatis Plus & Spring Boot ORM For Employee Class**

|  |
| --- |
| **package com.wes.goddard.controller;  import com.baomidou.mybatisplus.core.conditions.query.LambdaQueryWrapper; import com.baomidou.mybatisplus.extension.plugins.pagination.Page; import com.wes.goddard.common.R; import com.wes.goddard.entity.Employee; import com.wes.goddard.service.EmployeeService; import lombok.extern.slf4j.Slf4j; import org.apache.commons.lang.StringUtils; import org.springframework.beans.factory.annotation.Autowired; import org.springframework.util.DigestUtils; import org.springframework.web.bind.annotation.\*;  import javax.servlet.http.HttpServletRequest; import java.time.LocalDateTime;  @Slf4j @RestController @RequestMapping("/employee") public class EmployeeController {   @Autowired  private EmployeeService employeeService;   /\*\*  \* Add Employee Registration  \* @param request  \* @param employee  \* @return  \*/  @PostMapping("/Register")  public R<Employee> register(HttpServletRequest request,@RequestBody Employee employee){   //1. Encrypt the password submitted from the webpage with md5  String password = employee.getPassword();  password = DigestUtils.md5DigestAsHex(password.getBytes());   //2. Query the database for the submitted username  LambdaQueryWrapper<Employee> queryWrapper = new LambdaQueryWrapper<>();  queryWrapper.eq(Employee::getUsername,employee.getUsername());  Employee emp = employeeService.getOne(queryWrapper);   //3. If username not found, return Register fail  if(emp == null){  return R.error("Register Failed");  }   //4. Compare password, if not equal, return Register fail  if(!emp.getPassword().equals(password)){  return R.error("Register Failed");  }   //5. Check employee status, if disabled, return disabled  if(emp.getStatus() == 0){  return R.error("Account Disabled");  }   //6. register success, save the employee's ID to Session, and return register success  request.getSession().setAttribute("employee",emp.getId());  return R.success(emp);  }   /\*\*  \* employee Deletion  \* @param request  \* @return  \*  \*  \* when logged in, there will be a record in the session/local storage if press F12 on the webpage  \* when logged out, that record will be removed (might need to hit refresh)  \*/  @PostMapping("/deletion")  public R<String> logout(HttpServletRequest request){  //clear the  request.getSession().removeAttribute("employee");  return R.success("Logout Success");  }   /\*\*  \* Add employee  \* @param employee  \* @return  \*/  @PostMapping  public R<String> save(HttpServletRequest request,@RequestBody Employee employee){  log.info("Add employee，new employee info：{}",employee.toString());   //set initial default password 123456，and use md5 to encrpyt this initial password  employee.setPassword(DigestUtils.md5DigestAsHex("123456".getBytes()));  // employee.setCreateTime(LocalDateTime.now()); // employee.setUpdateTime(LocalDateTime.now()); // // //get current user (creater) id // Long empId = (Long) request.getSession().getAttribute("employee"); // // employee.setCreateUser(empId); // employee.setUpdateUser(empId);   employeeService.save(employee);   return R.success("Add new employee success");  }   /\*\*  \* Query employee info paginated  \* @param page  \* @param pageSize  \* @param name  \* @return  \*/  @GetMapping("/page")  public R<Page> page(int page, int pageSize, String name){  log.info("page = {},pageSize = {},name = {}" ,page,pageSize,name);   Page pageInfo = new Page(page,pageSize);   LambdaQueryWrapper<Employee> queryWrapper = new LambdaQueryWrapper();   queryWrapper.like(StringUtils.isNotEmpty(name),Employee::getName,name);   queryWrapper.orderByDesc(Employee::getUpdateTime);   employeeService.page(pageInfo,queryWrapper);   return R.success(pageInfo);  }   /\*\*  \* Edit Employee info based on ID  \*  \* Such as disabling/enabling accounts  \*  \* Workflow:  \* 1. when clicking disabling/enabling, it triggers statusHandle method  \* 2. in statusHandle, it starts ajax request via enableOrDisable, sending id and status (1 if currently 0, vice versa)  \* 3. controller receiving the data as encapsulated, and use MybatisPlus method to update  \*  \* @param employee  \* @return  \*/  @PutMapping  public R<String> update(HttpServletRequest request,@RequestBody Employee employee){  log.info(employee.toString());  // Long empId = (Long)request.getSession().getAttribute("employee"); // employee.setUpdateTime(LocalDateTime.now()); // employee.setUpdateUser(empId);  employeeService.updateById(employee);   return R.success("User update Success");  }   /\*\*  \* Query User info based on ID  \* @param id  \* @return  \*/  @GetMapping("/{id}")  public R<Employee> getById(@PathVariable Long id){  log.info("Querying employee info with ID ...");  Employee employee = employeeService.getById(id);  if(employee != null){  return R.success(employee);  }  return R.error("Employee not found");  } }** |