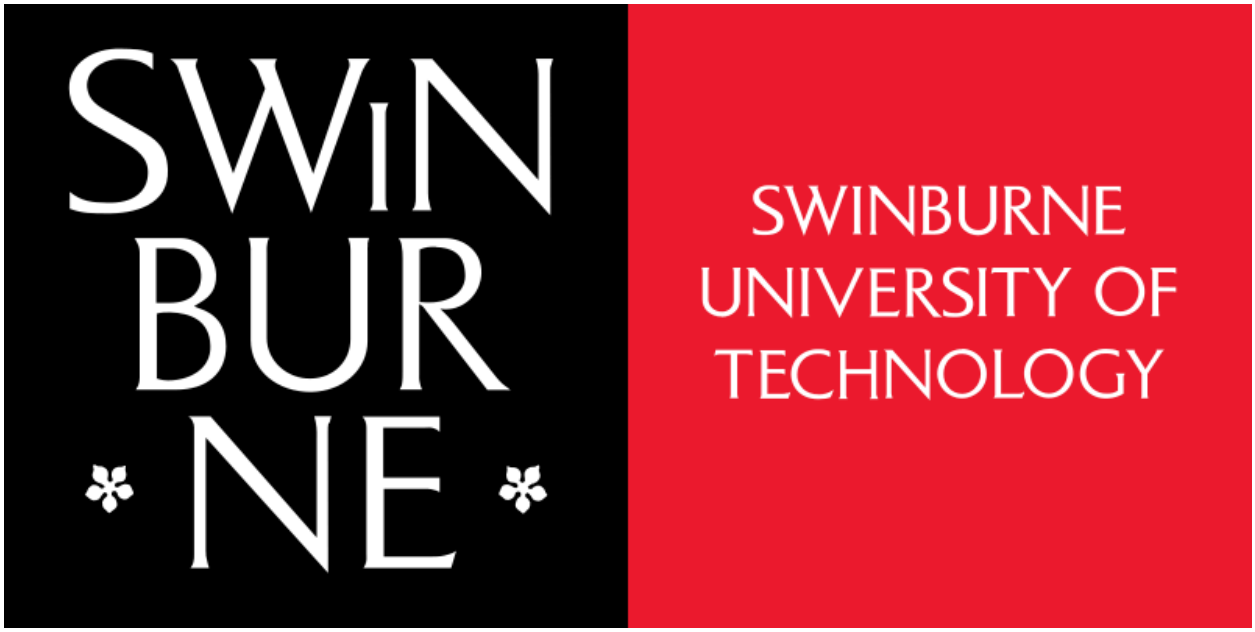

AI-based Project

Assignment 3 – COS30049

Group 81 – Fantastic Realtors

Names and Student IDS:

Anthony Ngo – 104548672

Chayan Kapoor – 104202680

Dave Nguyen – 104697710

Table of Contents this page

Introduction

The AI-powered house price prediction project in Melbourne seeks to create a strong machine-learning model that gives accurate insights for potential homeowners, investors, and anybody interested in the real estate market. Using real-world data from the Melbourne housing market, the project aims to discover critical variables that impact property values and utilise powerful machine-learning algorithms to provide accurate price projections. The following report gives insight into how the AI works on the website that we have created.

System Architecture

The System uses both React as a frontend and FastAPI as a backend, when combined it creates a property price prediction tool that is responsive. The user can input property details on the frontend, which then sends a request to the backend. To get prediction results the backend uses either Random Forest or Gradient Boosting model to process this data. Users gain insights on property values thanks to the models being trained to estimate prices based on factors like suburb, rooms, and property type.

Front-End Implementation:

This website intends to give clients an interactive tool for predicting house values utilising property information. It contains numerous key elements, namely a form for user input, data visualisation for enhanced insights, validation for data precision, and interaction with a back-end system for artificial intelligence forecasts.

This program consists of three key components: an interactive form for entering data, a chart for visualising predictions, and a method for receiving live data from the backend. I'll discuss how these function, with an emphasis on ActualVsPredictedChart.js and its role in gathering and showing data depending on user input.

User Input Form and Data Retrieval:

The user form is handled by PredictionForm.js, where users may enter particular property information to generate a home price estimate. When users submit the form, the data is quickly validated before being sent to the server via Axios (a popular promise-based HTTP client in React). Axios simplifies the management of asynchronous requests, which is really useful in this situation. In ActualVsPredictedChart.js, we employ a similar design in which user input drives the content. The drop down allows users to select a suburb. Every time customers pick a new one, the app retrieves the most recent actual and forecasted pricing data for that area, keeping

the presented chart up to current. `const [selectedSuburb, setSelectedSuburb] = useState('');` – This line creates a state variable to store whichever suburb the user selects.

```
const ActualVsPredictedChart = () => {
    const [suburbs, setSuburbs] = useState([]);             // List of suburbs
    const [selectedSuburb, setSelectedSuburb] = useState(''); // Selected suburb
    const [chartData, setChartData] = useState(null);
    const [error, setError] = useState(null);
```

Fig.1: Variable to store whichever suburb the user selects.

useEffect(() => {... }, [selectedSuburb]);- This React plugin detects updates in the selectedSuburb. Whenever the user selects a different suburb, the hook initiates a data retrieve, ensuring that the chart is updated to reflect the new location.

```
// Fetch all available suburbs on component mount
useEffect(() => {
    const fetchSuburbs = async () => {
        try {
            const response = await axios.get('http://127.0.0.1:8000/suburbs');
            setSuburbs(response.data.suburbs);
        } catch (error) {
            console.error("Error fetching suburbs:", error);
            setError("Failed to load suburbs. Please check the server.");
        }
    };
    fetchSuburbs();
}, []);
```

Fig.2: Fetch all available suburbs on component mount.

Data Visualization and Error Handling

The major element of this app's visualisation is a "Actual vs. Predicted Price" chart generated with the Line component of Chart.js (a popular data visualisation framework). This graphic refreshes dynamically based on the suburb selected by the user, allowing them to quickly view updated real vs. forecasted pricing statistics. ‘chartData’ is a state variable that contains the data to be shown, divided into two datasets: real prices and forecasted prices. Each dataset has its own colour and style to make it easier to identify.

axios.get(\http://127.0.0.1:8000/actual-vs-predicted/${selectedSuburb})' - This line makes an HTTP GET request to the server to retrieve data for the specified suburb. The server delivers arrays of actual and forecasted prices, which are then shown on the chart.

```
const response = await axios.get(`http://127.0.0.1:8000/actual-vs-predicted/${selectedSuburb}`);
const { actual, predicted } = response.data;
```

Fig.3: axios request to the server.

setError("Failed to load actual vs predicted data."); - Simple error handling. If there is a difficulty receiving data (for example, the server is not responding), the app displays an error message to notify users, so they can avoid questioning whether the data fails to show up.

```
} catch (error) {
    console.error("Error fetching actual vs predicted data:", error);
    setError("Failed to load actual vs predicted data.");
}
```

Fig.4: error message to notify users.

Back-End Integration and Real-Time Updates

The app interfaces with the backend via Axios, which allows it to transmit and receive data without refreshing the page. By using the selected suburb as a dynamic state variable, the app ensures that the data on the screen corresponds to the suburb the user selected. This approach enables a dynamic user experience by displaying forecasts and data in real time, delivering rapid visual feedback via the chart.

This flow allows users to simply study the data, change forecasts in real time, and interact naturally with the form and chart. Overall, it's a simple approach for people to get into home market projections and see outcomes immediately.

Component and App Files

Each visualization component (`ActualVsPredictedChart.js`, `AveragePriceByYearChart.js`, `PriceDistributionBySuburb.js`, and `PriceDistributionChart.js`) is modular, handling distinct parts of the user experience and promoting code reusability. The main application file, `App.js`, integrates these components, managing the application's state and data flow. By using React's `useState` and `useEffect` hooks, `App.js` ensures seamless interaction across form submissions, data visualizations, and predictions.

Index Files (index.js and index.css)

The `index.js` file serves as the entry point, rendering the React application into the DOM and initializing essential libraries. Meanwhile, `index.css` defines global styling, maintaining a consistent look across all pages. This separation of style and functionality allows for easy

updates and modular styling, ensuring the application remains visually cohesive (Heitkötter et al., 2022).

By combining user input, data visualisation, and real-time prediction capabilities, the application provides a comprehensive, user-friendly approach to exploring housing prices.

Back-End Implementation

FastAPI Server setup:

The server's core setup is housed in `main.py`, where the FastAPI server is initialised with: 'app = FastAPI()'. The file is also the main file for configuring routes and HTTPS requests. An essential part of the server setup revolves around Cross-Origin Resource Sharing (CORS middleware), which allows the API to interact with the frontend client host on a different domain, Without CORS middleware, cross-domain requests would be blocked by default for security reasons:

```python
# Initialize FastAPI
app = FastAPI()
app.add_middleware(
    CORSMiddleware,
    allow_origins=["http://localhost:3000", "http://127.0.0.1:3000"],
    allow_credentials=True,
    allow_methods=["*"],
    allow_headers=["*"],
)
```

Fig 5. initialising the FastAPI server and using CORSMiddleware.

This enables smooth communication with the front end now that all HTTP methods and headers are allowed. Once you start up the main.py it loads models and preprocessing tools such as a scaler and an encoder using joblib which is useful for handling large data objects.

```python
# Load trained model, encoder, and scaler
forest_model = joblib.load('forest_model.pkl')
encoder = joblib.load('encoder.pkl')
scaler = joblib.load('scaler.pkl')
gb_model = joblib.load('gb_model.pkl')
```

Fig 6. Loading models and tools

Once these models and tools are loaded the server wouldn't need to continuously reload the resources each time allowing for the server to respond to requests faster.

API endpoint documentation (Including request type, JSON response, etc.):

The server offers six endpoints which can be found within the `main.py` file. Each of these endpoints have a request type, and a JSON response with an actual value as an example instead of placeholders:

1. **Predict Property Price** (/predict)

```
@app.post("/predict")
```

Fig 7. post

Request type: POST

JSON response:

{

“prediction: "750000.75”

}

2. **Get Available Suburbs** (/suburbs)

```
# Endpoint to get available suburbs
@app.get("/suburbs")
```

Request type: GET

JSON response:

{

“suburbs”: “Abbotsford”,”Altona”,...

}

3. **Get Options for a Specific Suburb** (/options/{suburb})

```
# Endpoint to get options for a specific suburb
@app.get("/options/{suburb}")
```

Request type: GET

JSON response:

{

“rooms”: 2,

“types”: “h”,

“bathrooms”: 1,

“cars”: 1,

“regionname”: “Western Metropolitan”

}

4. **Get Price Distribution for a Suburb** (/price-distribution/{suburb})

```
@app.get("/price-distribution/{suburb}")
```

Request type: GET

JSON response:

{

“bin_edges”: 20000,40000, ...

“price_counts”: 10,20,...

“average_price”: 650000.50,...

“min_price”: 200000.00,
“max_price”: 960000.00,
}

5. **Actual vs Predicted Prices by Suburb** (/actual-vs-predicted/{suburb})

```
# Actual vs. Predicted Endpoint (Filtered by Suburb Only)
@app.get("/actual-vs-predicted/{suburb}")
```

Request type: GET

JSON response:
{
“actual”: 500000, 620000,...
“predicted”: 510000,615000,...
}

6. **Average Price by Year for a Suburb** (/average-price-by-year/{suburb})

```
@app.get("/average-price-by-year/{suburb}")
```

Request type: GET

JSON response:
{
“year_built”: 1990,1995,...
“average_price”: 350000, 420000,...
}

Robust Error Handling

Error Handling is a critical part of making this API user-friendly. We do this by making use of try-except blocks to catch any issues and return informative error messages to distinguish if they work. If a request fails to process due to any reason such as a missing value or potential issue with the dataset it’ll return a detailed message which helps users understand the issue.

This approach of error handling ensures users understand and get clear feedback on any issues. This helps when users are trying to diagnose the problem as they wouldn’t just see a generic error, instead they’re provided with a message that describes the problem which leads to faster troubleshooting.

AI Model Integration

In Assignment 2, we developed a predictive model for housing prices in Melbourne using `RandomForestRegressor` and `GradientBoostingRegressor`. Our model leveraged a range of property features, such as suburb, room count, property type, and location coordinates, to predict the property price. The following summarises the key steps in the initial model

development and highlights the modifications required to integrate this model into a web application.

Original Model Development in `assign2.py`

The original model workflow involved several preprocessing steps and model training. Key aspects included:

1. **Data Cleaning and Preprocessing**: We handled missing values by filling them with the median for numerical columns and "Unknown" for categorical columns. Then, categorical variables (e.g., suburb, property type) were encoded using one-hot encoding, and numerical features were scaled between 0 and 1 using `MinMaxScaler`. This prepared the data for model training.
2. **Model Training**: We trained two models: `RandomForestRegressor` and `GradientBoostingRegressor`. Both models were evaluated using metrics such as Mean Squared Error (MSE) and R^2 to assess their accuracy on the test set.

Modifications for Web Deployment

To make the model accessible as a web service, we implemented a FastAPI-based backend, creating RESTful API endpoints. This required several adjustments to ensure the model's predictions were accurate and responsive for real-time user input.

1. **Persisting and Loading Models**: Instead of training the model each time, we saved the trained models, scaler, and encoder using `joblib`.[1] This reduced computation time, allowing the server to respond quickly to prediction requests by directly loading the pre-trained models:

```
# Save models and transformers for consistent use
joblib.dump(forest_model, 'forest_model.pkl')
joblib.dump(gb_model, 'gb_model.pkl')
joblib.dump(encoder, 'encoder.pkl')
joblib.dump(scaler, 'scaler.pkl')

# Load models at application start
forest_model = joblib.load('forest_model.pkl')
encoder = joblib.load('encoder.pkl')
scaler = joblib.load('scaler.pkl')
gb_model = joblib.load('gb_model.pkl')
```

Fig.7: Using joblib for models and transformers.

2. **Defining API Endpoints**: We created several endpoints to handle user interactions:
    - **`/predict`**: Accepts property details (e.g., suburb, rooms, type) and returns a predicted price. This endpoint takes user inputs, preprocesses them, and routes them to the chosen model for prediction.

- **Additional endpoints** for data exploration and visualisation, such as `/price-distribution`, `/actual-vs-predicted`, and `/average-price-by-year`, enable users to analyse the data interactively.

For example, the `/predict` endpoint processes user input by applying the same preprocessing transformations used during training:

3. **Handling Missing Inputs with Defaults**: To support real-time predictions, we added a helper function that fills missing values with default values based on common attributes in the dataset. This ensures predictions are made even if some fields are left blank by the user.

```python
# Function to fill missing features with default values
def fill_missing_features(data):
    defaults = {
        "Landsize": 500.0,
        "BuildingArea": 150.0,
        "YearBuilt": 2000,
        "Car": 1,
        "Lattitude": -37.8136,
        "Longtitude": 144.9631,
        "Distance": 5.0,
        "Regionname": "Northern Metropolitan"  # a common value from the dataset
    }
    for key, value in defaults.items():
        if key not in data or data[key] is None:
            data[key] = value
    return data
```

Fig.8: Handling Missing values from the form.

4. **Scalability and Consistency**: To ensure consistency between training and deployment, we reused the saved encoder and scaler, applying the same transformations to user input data as were applied during model training. This maintained data integrity and alignment with the model's expectations, avoiding potential errors in prediction due to inconsistencies in data formatting.[2[

```python
# Encode categorical features using the pre-trained encoder
encoded_input = encoder.transform(input_df[['Suburb', 'Type', 'Regionname']])
encoded_input_df = pd.DataFrame(encoded_input, columns=encoder.get_feature_names_out(['Suburb', 'Type', 'Regionname']))

# Concatenate numerical and encoded categorical features
input_df = pd.concat([input_df.drop(columns=['Suburb', 'Type', 'Regionname']), encoded_input_df], axis=1)

# Scale numerical columns to match training data
input_df[numerical_columns] = scaler.transform(input_df[numerical_columns])
```

Fig.9: Using the loaded encoder and scaler.

Conclusion

This solution allows users to use real-world data to make better informed decisions about property investments by predicting prices. This supports our target audience of buyers, sellers, and investors looking to navigate the housing market as it provides valuable insights into property trends. Future enhancements that could be implemented include expanding our

dataset to cover more locations and deepening filtering options, which would allow this solution to be a more insightful tool for users.

Word count: 2085 words.

References:

[1] scikit-learn. (n.d.). *9. Model persistence*. [online] Available at:

https://scikit-learn.org/stable/model_persistence.html.

[2] Herman, M. (n.d.). *Deploying and Hosting a Machine Learning Model with FastAPI and Heroku*. [online] testdriven.io. Available at: https://testdriven.io/blog/fastapi-machine-learning/ [Accessed 1 Nov. 2024].

[3] Mozilla Developer Network (MDN) (n.d.). Cross-Origin Resource Sharing (CORS). Available at: https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS (Accessed: 3 November 2024).

[4]Auth0 (2020). What Is CORS? A Beginner's Guide to Cross-Origin Resource Sharing. Available at: https://auth0.com/blog/cors-tutorial-a-guide-to-cross-origin-resource-sharing/ (Accessed: 4 November 2024).