CM3070-final-report

September 8, 2024

Deep Learning for Personalised Property Recommendations System: Data Collection and Model Development Using Public Datasets

Guillermo Olmos Ranalli

September 7, 2024

Contents

1 Introduction

1.1 Project Concept

The Personalised Property Recommendation System aims to assist potential homebuyers in the UK real estate market by providing tailored property recommendations based on individual preferences and financial situations. By integrating historical transaction data from HM Land Registry with current property listings from OnTheMarket, the system delivers customised property suggestions. This project involves developing and evaluating various deep learning models to determine the most effective approach for property recommendation, framed as a classification task.

1.2 Motivation

Navigating the real estate market can be particularly challenging for new generations, including Millennials, Gen Z, and Gen Alpha, who face unique financial and lifestyle constraints. First-time buyers often struggle to find properties that meet their specific criteria within their budget. The abundance of property options and the complexity of property features necessitate a tool that can streamline the search process and offer personalised recommendations. This project seeks to address this need by developing a robust deep learning-based recommendation system.

1.3 Project Template

This project, based on "Project Idea Title 1: Deep Learning on a Public Dataset" from CM3015, uses publicly accessible real estate data from HM Land Registry and OnTheMarket to develop and compare deep learning models for personalised property recommendations. Following the methodology in "Deep Learning with Python" by F. Chollet, the prototype aims to improve model performance.

This project builds upon my previous work in CM2015, where I developed data collection scripts and utilised similar datasets. The current project expands this foundation by incorporating advanced machine learning for better recommendations.

1.4 Scope and Limitations

This prototype analyses various types of residential properties in Buckinghamshire, excluding shared ownership, retirement homes, new builds, auction listings, farms/land, park homes, and properties over £650,000. These filters were applied to data collected from OnTheMarket.com.

The prototype has limitations, including reliance on online data accuracy and availability. It does not cover commercial real estate or the rental market in Buckinghamshire.

1.5 Data Sources and Selection

1.5.1 Data Requirements

The primary goal of this prototype is to validate the feasibility of using machine learning techniques to predict property prices and recommend properties in the Buckinghamshire market. To achieve this, two main data sources were utilised: - **HM Land Registry Data**: This dataset provides comprehensive information on property sales in Buckinghamshire, including sale prices and transaction dates. - **Web Scraped Data from OnTheMarket.com**: A dataset comprising current property listings in Buckinghamshire, including asking prices, property types, and other relevant details.

To comprehensively analyse Buckinghamshire's property market, data from diverse sources were gathered: - **HM Land Registry Data**: This dataset provides comprehensive information on property sales in Buckinghamshire, including sale prices and transaction dates. - **Real Estate Listings**: Scraped data from OnTheMarket.com helps in understanding the ongoing trends and fluctuations in property prices and demands within Buckinghamshire. Python scripts were developed, organised in the `src` folder of this prototype, to efficiently collect and process the data, ensuring a robust and reliable dataset for analysis.

1.5.2 Choice of Data Sources

- **HM Land Registry Data**: Chosen for its reliability and comprehensive coverage of actual property sales in the UK. Accessed via gov.uk, under the Open Government Licence v3.0.
- **OnTheMarket.com**: Chosen as a current and active source for property listings, offering a real-time perspective on the market. Data was scraped in compliance with the site's terms and conditions, focusing on properties listed for sale in Buckinghamshire.

1.5.3 Methodology for Data Collection and Processing

- Data from HM Land Registry was downloaded in CSV format, covering transactions for the year 2023.
- Web scraping was conducted on OnTheMarket.com using Python scripts, focusing on gathering current listings in Buckinghamshire. The scraping process adhered to the website's robots.txt file and was conducted using a unique user agent.

1.5.4 Limitations and Constraints

- **Timeframe**: The HM Land Registry data covers only sales within 2023, and the scraped data reflects listings at the time of scraping. This temporal limitation means the analysis might not fully capture long-term market trends.
- **Geographical Scope**: The focus on Buckinghamshire alone may not provide a complete picture of broader regional or national property market trends.
- **Data Completeness**: While the Land Registry data is comprehensive for sales, the scraped data from OnTheMarket.com might not capture every property listing in the region, leading to potential gaps in the dataset.

1.6 Ethical Considerations

1.6.1 Data Sources and Permissions

HM Land Registry Data - HM Land Registry data is used under the Open Government Licence v3.0. - Proper attribution has been given as per the OGL requirements: "Contains HM Land Registry data © Crown copyright and database right 2021. This data is licensed under the Open Government Licence v3.0."

OnTheMarket.com Data - OnTheMarket.com data was collected via web scraping, strictly adhering to their robots.txt file, using a unique user agent with contact information, and employing rate limiting to respect their servers.

This prototype focuses on objective real estate data, avoiding personal judgements or assumptions. It aims to maintain neutrality, preventing negative impacts like market manipulation. Any personal data has been anonymized to protect privacy.

word count: 837

2 Chapter 2: A Literature Review

2.1 Introduction

In the realm of real estate, accurate property price prediction and personalised recommendation systems are crucial for assisting potential homebuyers and real estate professionals. The intersection of machine learning and real estate has gained substantial attention, with various methodologies explored to enhance prediction accuracy and personalisation. This literature review critically examines existing research on housing price prediction models and personalised recommendation systems, highlighting their methodologies, strengths, and limitations.

2.2 Housing Price Prediction Models

2.2.1 Hedonic-Based Regression Approaches

Historically, hedonic-based regression models have been utilised to determine the impact of various housing attributes on property prices. These models estimate prices based on factors such as location, size, and age of the property. Despite their widespread use, hedonic models face limitations such as difficulty in capturing nonlinear relationships and the need for extensive data preprocessing to handle heteroscedasticity and multicollinearity issues [1].

2.2.2 Machine Learning Techniques

The advent of machine learning has provided more sophisticated tools for housing price prediction, capable of handling complex, non-linear relationships between variables. The study by Park and Bae (2020) investigates the application of several machine learning algorithms to predict housing prices using data from Fairfax County, Virginia. The algorithms examined include C4.5, RIPPER, Naïve Bayesian, and AdaBoost [5].

- **C4.5 Algorithm**: This algorithm is an extension of the earlier ID3 algorithm and generates a decision tree used for classification purposes. In the context of housing price prediction, the decision tree helps identify the most significant variables influencing prices [6].
- **RIPPER Algorithm**: This is a rule-based learning algorithm that generates a set of rules to classify data. According to Park and Bae (2020), the RIPPER algorithm demonstrated superior performance in terms of accuracy compared to other models tested in their study [5].
- **Naïve Bayesian**: This probabilistic classifier is based on Bayes' theorem and assumes independence between predictors. Despite its simplicity, it can be effective for certain types of classification problems [2].
- **AdaBoost**: This ensemble method combines multiple weak classifiers to create a strong classifier. It adjusts the weights of misclassified instances, thereby improving the model's accuracy over successive iterations [3].

Park and Bae's study concludes that the RIPPER algorithm consistently outperformed the other models in terms of classification accuracy for housing price prediction. This finding is significant as it highlights the potential of rule-based algorithms in capturing the complexities of housing market data [5].

2.3 Content-Based Recommender Systems

Content-based recommender systems are crucial for providing personalised suggestions based on user preferences and item attributes. Lops, Gemmis, and Semeraro (2011) provide a comprehensive overview of the state-of-the-art techniques and trends in content-based recommendation systems [4].

- **Feature Extraction**: Content-based systems rely heavily on extracting meaningful features from items. In the context of real estate, features such as property type, location, price, and amenities are essential.
- **Similarity Calculation**: These systems calculate the similarity between items based on their features. For real estate, properties with similar attributes (e.g., location, price range) are considered similar and thus recommended to users with matching preferences.
- **User Profiles**: Content-based systems maintain profiles for users, capturing their preferences and interaction history. This allows the system to tailor recommendations based on individual user needs.

Lops et al. (2011) highlight the challenges in content-based recommender systems, such as the cold start problem, where new users or items lack sufficient data for effective recommendations. However, integrating advanced machine learning techniques can mitigate some of these issues by improving feature extraction and similarity calculations [4].

2.4 Discussion

The research conducted by Park and Bae (2020) underscores the importance of selecting appropriate machine learning algorithms for housing price prediction. Their comparative analysis provides valuable insights into the strengths and weaknesses of different models. For instance, while ensemble methods like AdaBoost are generally robust, rule-based algorithms such as RIPPER can offer higher accuracy for specific datasets [5].

This study also emphasizes the need for comprehensive data preprocessing, including the selection of relevant features and handling missing values, to enhance the predictive performance of machine learning models. Additionally, the integration of various algorithms can potentially lead to the development of a hybrid model that leverages the strengths of each approach [5].

2.5 Conclusion

The literature on housing price prediction demonstrates that machine learning techniques, particularly rule-based algorithms like RIPPER, can significantly improve the accuracy of price predictions. The study by Park and Bae (2020) serves as a critical reference point for developing advanced models that can aid real estate stakeholders in making informed decisions [5].

Further research should focus on integrating these models with personalised recommendation systems to provide comprehensive solutions for real estate buyers and sellers. By leveraging machine learning's capabilities, the real estate industry can enhance its analytical tools, leading to more accurate and reliable property valuations.

Word count chapter 2: 787

3 Chapter 3: A Design

3.1 Project Overview

The Personalized Property Recommendation System aims to integrate historical transaction data from HM Land Registry and current property listings from OnTheMarket to provide tailored property recommendations based on user preferences and financial situations. The project follows "Project Idea Title 1: Deep Learning on a Public Dataset" and aims to find the most effective model for property price prediction and recommendation using deep learning techniques.

3.2 Domain and Users

3.2.1 Domain

The project is situated in the real estate domain, focusing on residential properties in the UK. It leverages publicly available datasets to build a recommendation system that aids potential homebuyers in making informed decisions.

3.2.2 Users

The primary users of the system are: - **First-time homebuyers**: Individuals looking for their first property purchase who need tailored recommendations based on their budget and preferences. - **Real estate agents**: Professionals who can use the system to provide clients with data-driven property suggestions. - **Property investors**: Individuals or companies looking to invest in real estate who require accurate property price predictions and recommendations.

3.3 Justification of Design Choices

3.3.1 User Needs

The design choices are informed by the needs of users in the real estate market: - **Personalization**: Users require personalized property recommendations that match their financial constraints and preferences. - **Accurate Predictions**: Accurate property price predictions help users make informed decisions. - **Usability**: The system must be easy to use and provide quick, relevant recommendations.

3.3.2 Domain Requirements

The real estate domain requires: - **Integration of Diverse Data Sources**: Combining historical transaction data with current listings to provide a comprehensive view. - **Handling Non-linear Relationships**: Using advanced machine learning models to capture complex patterns in the data.

3.4 Project Structure

3.4.1 Data Collection and Preprocessing

- **Data Sources**: Historical transaction data from HM Land Registry and current property listings from OnTheMarket.
- **Web Scraping**: Scripts to collect real-time data from OnTheMarket.
- **Data Cleaning**: Handling missing values, standardizing formats, and filtering relevant data.
- **Data Integration**: Merging datasets to create a unified data source.

3.4.2 Model Development

- **Feature Selection**: Identifying relevant features such as price, location, property type, etc.
- **Model Selection**: Experimenting with various machine learning algorithms (e.g., C4.5, RIPPER, Naïve Bayesian, AdaBoost) to identify the best-performing model.
- **Training and Validation**: Splitting the data into training and validation sets, training the model, and evaluating its performance.

3.4.3 Recommendation System

- **User Profile Creation**: Collecting user preferences and financial information.
- **Similarity Calculation**: Using content-based filtering to match properties with user profiles.
- **Property Ranking**: Ranking properties based on their relevance to the user's preferences and budget.
- **Feedback Loop**: Incorporating synthetic user feedback to continuously improve the recommendation system.

3.5 Technologies and Methods

3.5.1 Technologies

- **Python**: Primary programming language for data processing and model development.
- **TensorFlow and Keras**: Libraries for building and training deep learning models.
- **Pandas and NumPy**: Libraries for data manipulation and analysis.
- **Scikit-Learn**: Library for implementing various machine learning algorithms and evaluation metrics.
- **BeautifulSoup and Requests**: Libraries for web scraping.
- **Matplotlib**: Library for data visualization.
- **Node.js and Express.js**: For building the backend of the web application.
- **HTML, CSS, and JavaScript**: For developing the frontend of the web application.

3.5.2 Methods

- **Data Preprocessing**: Cleaning and integrating data from multiple sources.
- **Machine Learning**: Developing and comparing different machine learning models to identify the most effective one for price prediction.
- **Content-Based Filtering**: Creating a recommendation system based on the features of the properties and user preferences.
- **Evaluation Metrics**: Using Mean Absolute Error (MAE) and Root Mean Squared Error (RMSE) to evaluate model performance.

3.6 Work Plan

3.6.1 Major Tasks and Timeline

- **Data Collection and Preprocessing (Weeks 1-4)**
  - Collect data from HM Land Registry and OnTheMarket
  - Clean and preprocess data
  - Integrate datasets
- **Model Development (Weeks 5-10)**
  - Feature selection
  - Develop and train machine learning models
  - Evaluate models using MAE and RMSE
- **Recommendation System Development (Weeks 11-14)**
  - Develop content-based filtering system
  - Integrate price prediction with user preferences
- **Web Interface Development (Week 11)**
  - Develop a user-friendly web interface for the recommendation system
- **Docker Setup and Integration (Week 12)**
  - Set up Docker to orchestrate the entire system
- **Experiment with Additional Models (Weeks 13-14)**
  - Implement and compare models like Gradient Boosting, Random Forests, and advanced deep learning architectures
  - Optimize hyperparameters for each model and compare their performance using various metrics
- **Expand Features and Incorporate New Parameters (Week 15)**
  - Collect and preprocess additional data for new features (e.g., socioeconomic and environmental factors)
  - Perform feature engineering to create new features from the existing data
  - Integrate these features into the model and evaluate their impact on performance
- **Geographical Expansion (Week 16)**
  - Collect data for regions beyond Buckinghamshire
  - Preprocess and integrate this data into the existing dataset
  - Evaluate the model's performance on the expanded dataset
- **User-Centric Testing and Feedback Collection (Week 17)**
  - Design a user-friendly interface for inputting preferences and receiving recommendations
  - Simulate user interactions using synthetic data
  - Collect feedback through synthetic data to refine the system
- **Replicate and Compare with High-Quality Models (Week 18)**
  - Replicate models from high-quality published papers
  - Compare their performance with your models and analyze the differences
- **Final Model Tuning and Evaluation (Week 19)**
  - Fine-tune the best-performing models
  - Conduct a thorough error analysis and identify areas for further improvement
  - Finalize the model and prepare it for deployment
- **Report Writing & Finalization (Week 20)**
  - Document all findings, methodologies, and results
  - Ensure the report is well-structured, with clear explanations and justifications for each step

– Prepare for submission, ensuring all requirements are met

Week	Task
1-4	Data Collection & Preprocessing
5-10	Model Development
11	Web Interface Development
12	Docker Setup and Integration
13-14	Experiment with Additional Models
15	Expand Features and Incorporate New Parameters
16	Geographical Expansion (Collect Data for New Areas)
17	User-Centric Testing and Feedback Collection
18	Replicate and Compare with High-Quality Models
19	Final Model Tuning and Evaluation
20	Report Writing & Finalization

3.7 Testing and Evaluation Plan

3.7.1 Testing

- **Unit Testing**: Test individual components (e.g., data collection scripts, model training functions) to ensure they work as expected.
- **Integration Testing**: Ensure that different components (e.g., data integration, model prediction, recommendation system) work together seamlessly.
- **Synthetic User Testing**: Simulate user interactions using synthetic data to evaluate the recommendation system's usability and effectiveness.

3.7.2 Evaluation

- **Model Evaluation**: Use MAE and RMSE to evaluate the accuracy of the price prediction model.
- **Synthetic User Feedback**: Use synthetic data to simulate user feedback regarding the relevance and usefulness of the property recommendations.
- **Performance Metrics**: Track the system's performance in terms of response time, accuracy, and user satisfaction.

By following this structured approach and incorporating these components, the project aims to deliver a robust and effective personalized property recommendation system that meets user needs and leverages advanced machine learning techniques.

4 Chapter 4: Implementation

4.1 System Architecture Overview

The UK Real Estate Recommendation System is designed with a modular and scalable architecture to efficiently collect, process, and analyze property data for personalized recommendations. The system architecture, as illustrated in Figure 4.1, consists of several key components:

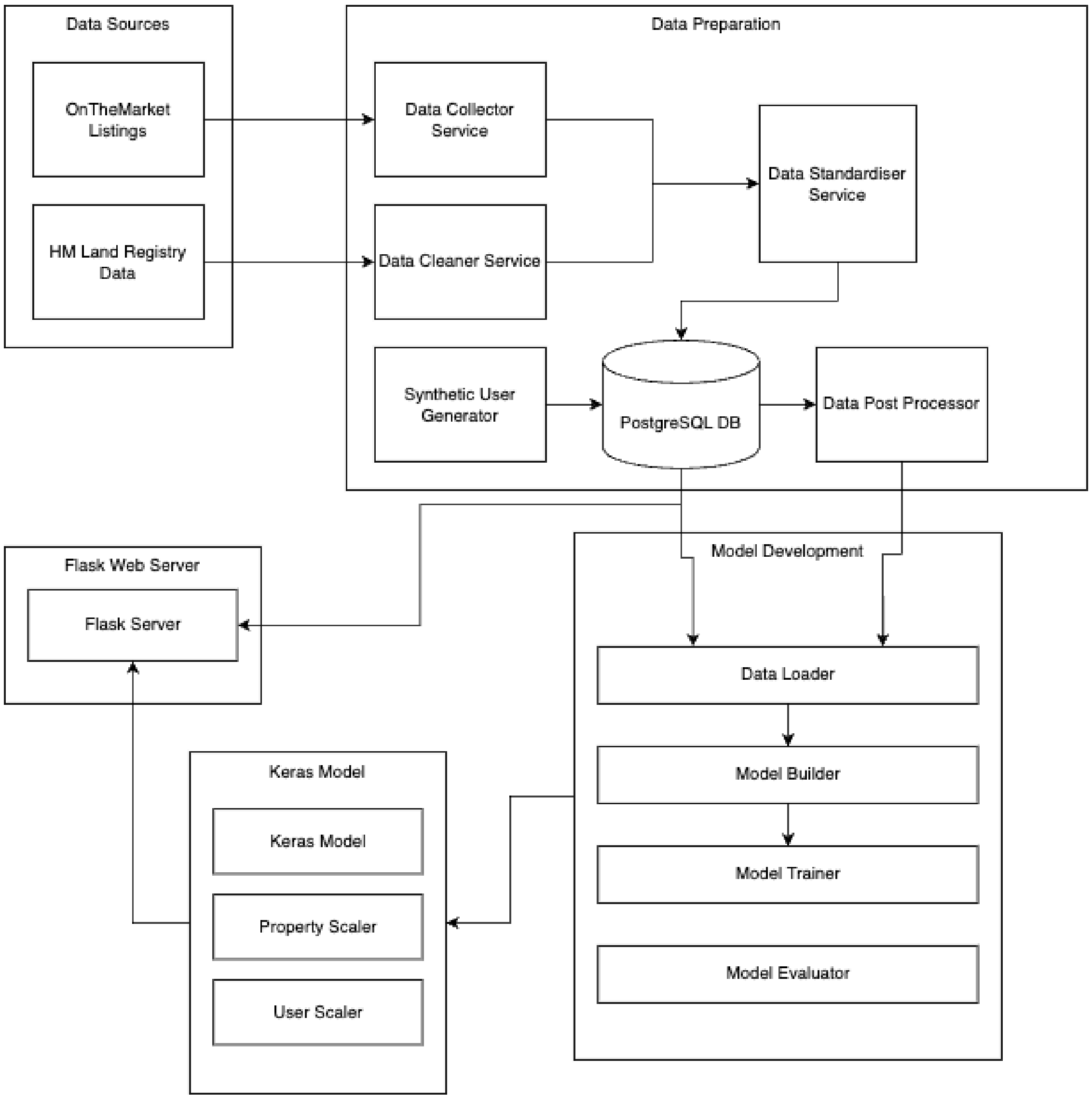


Figure 4.1: System Architecture Diagram

4.1.1 Data Sources

1. **OnTheMarket Listings:** Current property listings scraped from the OnTheMarket website, covering multiple shires including Buckinghamshire, Bedfordshire, Oxfordshire, Northamptonshire, Hertfordshire, and Berkshire.
2. **HM Land Registry Data:** Historical property transaction data from the UK government, providing comprehensive information for properties across multiple counties, including Buckinghamshire, Bedford, Oxfordshire, North Northamptonshire, West Northamptonshire, Hertfordshire, and West Berkshire. This data includes sale prices, property types, and locations.

4.1.2 Data Preparation

1. **Data Collector Service:** Web scrapes current property listings from OnTheMarket.com for multiple shires, using price segmentation to ensure comprehensive coverage.
2. **Data Cleaner Service:** Processes and cleans the HM Land Registry data, filtering for specific shires and years, and standardizing county names.
3. **Data Standardiser Service:** Integrates and standardizes data from both sources.
4. **Synthetic User Generator:** Creates synthetic user profiles for testing and development purposes.

4.1.3 Data Storage

1. **PostgreSQL Database:** Centralized storage for all processed and standardized data.

4.1.4 Data Processing

1. **Data Post Processor:** Performs additional data transformations and feature engineering.

4.1.5 Model Development

1. **Data Loader:** Retrieves and prepares data for model training.
2. **Model Builder:** Constructs the neural network architecture.
3. **Model Trainer:** Trains the model on the prepared dataset.
4. **Model Evaluator:** Assesses the model's performance and generates evaluation metrics.

4.1.6 Web Application

1. **Flask Web Server:** Hosts the user interface and handles user requests.
2. **Keras Model:** The trained neural network model for generating property recommendations.
3. **Property Scaler and User Scaler:** Normalize input data for consistent model predictions.

This architecture ensures a streamlined flow of data from collection to recommendation, with each component designed to handle specific tasks in the pipeline. The modular design allows for easy maintenance, updates, and scalability of individual components without affecting the entire system.

4.2 Data Collection and Data Cleaning

4.2.1 Web Scraping Methodology

To complement the historical data from HM Land Registry, we implemented a sophisticated web scraping solution for collecting current property listings from OnTheMarket.com across multiple shires. The process is modularised into four main components:

1. `robot_check.py`: Ensures compliance with the website's `robots.txt` file.
2. `crawler.py`: Discovers and collects property listing URLs using requests and BeautifulSoup, implementing rate limiting to avoid server overload.
3. `scraper.py`: Extracts specific data from web pages, including prices, addresses, and property features.
4. `data_collector_service.py`: Orchestrates the entire web scraping process, managing the workflow between the crawler and scraper modules. It implements the following key features:

- Multi-shire data collection: Iterates through each shire (Buckinghamshire, Bedfordshire, Oxfordshire, Northamptonshire, Hertfordshire, and Berkshire) to collect data separately.
- Price segmentation: Divides the property price range into segments (e.g., £100,000 increments) to ensure comprehensive coverage across all price brackets.
- Incremental data saving: Saves collected data for each shire separately, allowing for easier management and processing of large datasets.

Ethical Considerations Our web scraping adhered to ethical standards, including respecting `robots.txt`, using a unique user agent, implementing rate limiting, and ensuring non-disruptive interaction with OnTheMarket.com.

Note on Script Execution Time Due to our ethical scraping approach, the script's execution takes longer for larger datasets, particularly given the rate limits and crawl delays we adhere to. This ensures responsible scraping while avoiding potential blocking by the website.

We can run `data_collector_service.py` from the CLI or from the Jupyter Notebook using `%run` and provide the `max_price` argument to control the data collection scope. For this prototype, data was collected for properties priced up to £650,000. This data can be found in the `data` folder

[ ]:
```
# %run src/data-collector/data_collector_service.py 650000
%run src/data-collector/data_collector_service.py 120000
```

4.3 Standardisation and Preprocessing

The data cleaning and standardisation process is a crucial step in our pipeline, ensuring that data from different sources is consistent and ready for analysis. This process is handled by several specialized modules within the `data_standardiser` package.

4.3.1 Data Standardiser Architecture

The `data_standardiser` package is structured as follows:

- `address_utils.py`: Handles address normalization and cleaning.
- `constants.py`: Stores constant values used across the standardization process.
- `county_mapping.py`: Manages standardization of county names.
- `data_processing.py`: Contains core data processing and standardization functions.
- `database_operations.py`: Manages database interactions for data storage and retrieval.
- `geocoding.py`: Handles the geocoding process to add latitude and longitude data.
- `logging_config.py`: Configures logging for the standardization process.
- `main.py`: Orchestrates the entire standardization process.
- `property_utils.py`: Provides utility functions for property-specific data processing.
- `utils.py`: Contains general utility functions used across the package.

Modularity and Code Structure This modular approach to data standardization ensures that our data is consistent, accurate, and ready for the subsequent stages of analysis and model development. It also allows for easy maintenance and updates to individual components of the standardisation process as needed.

4.3.2 Key Standardisation Processes

1. **Address Normalization** The `address_utils.py` module provides functions to clean and normalize addresses, ensuring consistency across different data sources. This includes removing special characters, standardizing formatting, and handling common address variations.

2. **County Standardization** `county_mapping.py` is responsible for standardizing county names. This is crucial when dealing with data from multiple sources that might use different naming conventions for the same counties.

3. **Data Processing and Feature Extraction** `data_processing.py` contains core functions for processing raw data. This includes:

   - Standardizing price formats
   - Extracting and standardizing property features (e.g., number of bedrooms, bathrooms)
   - Standardizing property types
   - Handling date conversions

4. **Geocoding** The `geocoding.py` module implements a robust geocoding process using both Nominatim and ArcGIS services. It includes features like:

   - Caching to avoid redundant API calls
   - Error handling and retries
   - Fallback mechanisms when one service fails

5. **Database Operations** `database_operations.py` manages the interaction between the standardized data and the database. It handles:

   - Inserting and updating historical property data
   - Processing and inserting listing data
   - Merging data from different sources in the database

4.3.3 Standardization Workflow

The standardization process, orchestrated by `src/data_standardiser/main.py`, follows these general steps:

1. Load raw data from both HM Land Registry and scraped listings.
2. Clean and standardize addresses and county names.
3. Process and standardize property details (prices, types, features).
4. Perform geocoding to add latitude and longitude data.
5. Merge data from different sources, resolving conflicts and duplicates.
6. Store the standardized data in the database for further analysis.

4.4 Data Analysis and Feature Engineering

4.4.1 Initial Data Exploration and Analysis

Examination of the HM Land Registry Dataset We examined the UK-wide property transaction dataset (`pp-monthly-update-new-version.csv`) from HM Land Registry and filtered it to focus on transactions in Buckinghamshire.

[ ]:
```
import pandas as pd

# Path to your CSV file
file_path = './data/historical-data/pp-monthly-update-new-version.csv'

# Read the CSV file and display the first 10 rows
df = pd.read_csv(file_path)
print("First 10 rows of the data are:")
df.head(10)
```

4.4.2 Cleaning and Preparing Data

Data Cleaning Methodology Our data cleaning methodology, using tools like pandas, involves filtering, standardising, and handling missing/erroneous data from HM Land Registry and OnTheMarket.com for our Buckinghamshire property market analysis.

Implementing Data Cleaning with `data_cleanser_service.py` The `data_cleanser_service.py` script is a key component in refining raw HM Land Registry data for our Buckinghamshire property market analysis.

Key Features of `data_cleanser_service.py`:

- **Header Assignment**: The script assigns column headers to the HM Land Registry dataset (`pp-monthly-update-new-version.csv`) based on definitions from their website (https://www.gov.uk/guidance/about-the-price-paid-data).
- **Loading and Structuring Data**: Loads the CSV data into a `pandas` DataFrame.
- **Date Conversion and Filtering**: Retains only properties located in Buckinghamshire.

[ ]:
```
# data-cleanser/data_cleanser_service.py
import pandas as pd

def cleanse_data(input_file, output_file):
    # Define the headers based on the provided breakdown
    headers = ["Unique Transaction Identifier", "Price", "Date of Transaction",
               "Postal Code", "Property Type", "Old/New", "Duration",
               "PAON", "SAON", "Street", "Locality", "Town/City",
               "District", "County", "PPD Category Type", "Record Status"]

    # Load the CSV file without headers
    data = pd.read_csv(input_file, header=None, names=headers)

    # Convert Date of Transaction to datetime for filtering
    print("Converting dates and filtering data...")
    data['Date of Transaction'] = pd.to_datetime(data['Date of Transaction'])

    # Filter for properties in Buckinghamshire and from the year 2023
    data['Date of Transaction'] = pd.to_datetime(data['Date of Transaction'])
    filtered_data = data[(data['County'].str.upper() == 'BUCKINGHAMSHIRE') &
```

```
                      (data['Date of Transaction'].dt.year == 2023)]

    print(f"Number of records after filtering: {len(filtered_data)}")

    print("Saving cleaned data to CSV file...")
    # Save the cleaned data to a new CSV file
    filtered_data.to_csv(output_file, index=False)
    print("Data cleansing process completed successfully.")


# File paths
input_csv = './data/historical-data/pp-monthly-update-new-version.csv'  #␣
 ↪Update with actual path
output_csv = './data/historical-data/buckinghamshire_2023_cleaned_data.csv'  #␣
 ↪Update with desired output path

cleanse_data(input_csv, output_csv)
```

4.4.3 Enhancing Data with Geocoding and Merging using `data_standardiser_service.py`

`data_standardiser_service.py` enhances and merges datasets:

Key Features of `data_standardiser_service.py`: 1. **Geocoding**: Uses Nominatim and ArcGIS for rate-limited geocoding, with a fallback mechanism if one fails. 2. **Price Standardisation**: Converts various price formats into a uniform numerical format. 3. **Address Normalization**: Standardizes addresses for consistency. 4. **Dataset Merging**: Combines the processed scraped and registry data into a single DataFrame. 5. **Saving & Updating**: Saves the enriched dataset and updates existing data. 6. **Property Type Classification**: Ensures consistency across the dataset. 7. **Updating Existing Preprocessed Data**: Classifies scraped data rows according to the registry data classification system.

```
[ ]: # data-standardiser/data_standariser_service.py but with relative file path␣
      ↪changed
     import pandas as pd
     import os
     from datetime import datetime
     from geopy.geocoders import Nominatim, ArcGIS
     from geopy.extra.rate_limiter import RateLimiter
     from geopy.exc import GeocoderTimedOut, GeocoderQuotaExceeded
     import time

     # Initialize Nominatim API
     geolocator = Nominatim(user_agent="StudentDataProjectScraper/1.0 (Contact:␣
      ↪gor5@student.london.ac.uk)")
     geolocator_arcgis = ArcGIS(user_agent="StudentDataProjectScraper/1.0 (Contact:␣
      ↪gor5@student.london.ac.uk)")
```

```
# Rate limiter to avoid overloading the API
geocode = RateLimiter(geolocator.geocode, min_delay_seconds=2)
geocode_arcgis = RateLimiter(geolocator_arcgis.geocode, min_delay_seconds=2)

# Mapping for converting scraped property types to registry property types
scraped_to_registry_property_type_mapping = {
    'Apartment': 'F',
    'Barn conversion': 'O',
    'Block of apartments': 'F',
    'Bungalow': 'D',
    'Character property': 'O',
    'Cluster house': 'O',
    'Coach house': 'F',
    'Cottage': 'D',
    'Detached bungalow': 'D',
    'Detached house': 'D',
    'Duplex': 'F',
    'End of terrace house': 'T',
    'Equestrian property': 'O',
    'Farm house': 'O',
    'Flat': 'F',
    'Ground floor flat': 'F',
    'Ground floor maisonette': 'F',
    'House': 'D',
    'Link detached house': 'D',
    'Lodge': 'O',
    'Maisonette': 'F',
    'Mews': 'O',
    'Penthouse': 'F',
    'Semi-detached bungalow': 'D',
    'Semi-detached house': 'S',
    'Studio': 'F',
    'Terraced house': 'T',
    'Townhouse': 'D'
}

def geocode_address(address):
    try:
        location = geocode(address)
        if location:
            print(f"Geocoded '{address}': Latitude {location.latitude}, ␣
 ↪Longitude {location.longitude}")
            return location.latitude, location.longitude
        else:
            # Fallback to ArcGIS if Nominatim fails
            location = geocode_arcgis(address)
            if location:
                print(f"Geocoded '{address}': Latitude {location.latitude},␣
↪Longitude {location.longitude}")
                return location.latitude, location.longitude
            else:
                print(f"No result for '{address}'")
                return None, None
    except GeocoderQuotaExceeded:
        print("Quota exceeded for geocoding API")
        return None, None
    except GeocoderTimedOut:
        print("Geocoding API timed out")
        return None, None
    except Exception as e:
        print(f"Error geocoding {address}: {e}")
        return None, None

def check_preprocessed_file(file_path):
    """Check if the preprocessed file exists and has latitude and longitude."""
    if os.path.exists(file_path):
        df = pd.read_csv(file_path)
        if 'latitude' in df.columns and 'longitude' in df.columns:
            if df[['latitude', 'longitude']].notnull().all().all():
                # File exists and latitude and longitude are filled
                return True
    return False

def standardise_price(price):
    """
    Convert a price string to a numerical value.
    Handles strings like '£275,000' and converts them to 275000.
    """
    if not isinstance(price, str):
        return price  # If it's already a number, return as-is

    # Removing currency symbols and commas
    price = price.replace('£', '').replace(',', '').replace('€', '').strip()

    try:
        # Convert to float or int
        price_value = float(price) if '.' in price else int(price)
    except ValueError:
        # Handle cases where conversion fails
        print(f"Warning: Could not convert price '{price}' to a number.")
        price_value = None

    return price_value
```

```
def normalize_address_scraped(address):
    """
    Normalize addresses from the scraped data.
    """
    # Assuming the county is always 'Buckinghamshire' if not specified
    if 'Buckinghamshire' not in address:
        address += ', Buckinghamshire'
    return address.strip()

def normalize_address_land_registry(row):
    # Convert each component to a string to avoid TypeError
    components = [
        str(row['Street']),
        str(row['Locality']),
        str(row['Town/City']),
        str(row['District']),
        str(row['County'])
    ]
    # Join the non-empty components
    return ', '.join(filter(None, components))

# Read JSON, standardize price, normalize address, add source column

def read_and_process_scraped_data(scraped_file_path, skip_geocoding):
    # Read the scraped data
    scraped_data = pd.read_json(scraped_file_path)
    scraped_data['price'] = scraped_data['price'].apply(standardise_price)
    scraped_data['normalized_address'] = scraped_data['address'].
 ↪apply(normalize_address_scraped)
    scraped_data['source'] = 'scraped'

    if not skip_geocoding:
        lat_long = scraped_data['normalized_address'].apply(geocode_address)
        scraped_data['latitude'] = lat_long.apply(lambda x: x[0] if x else None)
        scraped_data['longitude'] = lat_long.apply(lambda x: x[1] if x else␣
 ↪None)

    return scraped_data

def read_and_process_registry_data(registry_file_path, skip_geocoding):
    registry_data = pd.read_csv(registry_file_path)
    registry_data['Price'] = registry_data['Price'].apply(standardise_price)
    registry_data['normalized_address'] = registry_data.
 ↪apply(normalize_address_land_registry, axis=1)
    registry_data.rename(columns={'Price': 'price'}, inplace=True)
    registry_data['source'] = 'registry'
    if not skip_geocoding:
        lat_long = registry_data['normalized_address'].apply(geocode_address)
        registry_data['latitude'] = lat_long.apply(lambda x: x[0] if x else␣
↪None)
        registry_data['longitude'] = lat_long.apply(lambda x: x[1] if x else␣
↪None)

    return registry_data

def update_date_column(df, source_column, new_date):
    """
    Update 'Date' column in the DataFrame based on source.
    """
    df['Date'] = pd.NaT
    df.loc[df['source'] == 'registry', 'Date'] = pd.
↪to_datetime(df[source_column])
    df.loc[df['source'] == 'scraped', 'Date'] = new_date
    return df

def merge_price_columns(df):
    """
    Merge 'price' and 'Price' columns and update 'Price' with 'price' values␣
↪for scraped data.
    """
    # Use 'price' from scraped data if it is not NaN, else use 'Price' from␣
↪registry data
    df['Price'] = df.apply(lambda x: x['price'] if pd.notna(x['price']) else␣
↪x['Price'], axis=1)
    return df

def apply_property_type_mapping(df):
    """
    Apply property type mapping to the DataFrame, only if 'property_type' is␣
↪not null.
    """
    # Apply mapping only where 'property_type' is not null
    mask = df['property_type'].notnull()
    df.loc[mask, 'Property Type'] = df.loc[mask, 'property_type'].
↪map(scraped_to_registry_property_type_mapping)
    return df

def process_and_save_data(scraped_data, registry_data, output_file_path):
    """
    Process and save merged data.
    """
    # Merge datasets
    merged_data = pd.concat([scraped_data, registry_data], ignore_index=True)

    # Update the date column
    merged_data = update_date_column(merged_data, 'Date of Transaction',␣
 ↪datetime(2023, 12, 31))

    # Apply property type mapping
    merged_data = apply_property_type_mapping(merged_data)

    # Merge 'price' and 'Price' columns
    merged_data = merge_price_columns(merged_data)

    # Save merged data
    merged_data.to_csv(output_file_path, index=False)
    print(f"Merged data saved successfully to '{output_file_path}'.")

def main():
    scraped_file = './data/property_data_650000.json'
    registry_file = './data/historical-data/buckinghamshire_2023_cleaned_data.
 ↪csv'
    output_file = './data/preprocessed-data/preprocessed.csv'

    if check_preprocessed_file(output_file):
        # Read existing preprocessed data
        preprocessed_data = pd.read_csv(output_file)
        print(f"Using existing preprocessed data from '{output_file}'.")

        # Update the date column in the existing data
        preprocessed_data = update_date_column(preprocessed_data, 'Date of␣
 ↪Transaction', datetime(2023, 12, 31))

        # Apply property type mapping
        preprocessed_data = apply_property_type_mapping(preprocessed_data)

        # Apply property type mapping and merge price columns
        preprocessed_data = merge_price_columns(preprocessed_data)

        # Save updated data
        preprocessed_data.to_csv(output_file, index=False)
        print(f"Updated data saved successfully to '{output_file}'.")
    else:
        # Process new data
        scraped_data = read_and_process_scraped_data(scraped_file, False)
        registry_data = read_and_process_registry_data(registry_file, False)

        # Process and save merged data
        process_and_save_data(scraped_data, registry_data, output_file)
    print("Data processing completed.")

if __name__ == "__main__":
    main()
```

The dataset below combines web-scraped data (December 2023) and HM Land Registry data (2023) for Buckinghamshire properties. The first and last five entries are shown, summarising the dataset's structure and content.

```
[ ]: import pandas as pd

     # Path to your CSV file
     file_path = './data/preprocessed-data/preprocessed.csv'

     # Read the CSV file
     df = pd.read_csv(file_path)

     # Display the first 5 rows
     print("First 5 rows of the data are:")
     print(df.head(5))

     # Display the last 5 rows
     print("\nLast 5 rows of the data are:")
     print(df.tail(5))

     # Print the column names
     print("Column names:")
     print(df.columns)
```

4.5 Feature Engineering

Feature engineering is a crucial step in our data preparation pipeline, enhancing our model's ability to capture relevant patterns and relationships in the data. Our feature engineering process, implemented primarily in `data_post_processor.py` and `location_classifier.py`, involves several key steps:

4.5.1 Temporal Features

We extract and transform date-related information: - `year`, `month`, and `day_of_week` from the transaction dates. - `days_since_date`: Calculates the number of days between the listing/transaction date and the current date. - `listing_recency`: A categorical feature binning the `days_since_date` into meaningful categories (e.g., 'Today', 'Last Week', 'Last 2 Weeks', etc.).

4.5.2 Affordability Metrics

We create several features to capture affordability: - `price_to_income_ratio`: Property price divided by average income. - `price_to_savings_ratio`: Property price divided by average savings. - `affordability_score`: A composite score based on income, savings, and property price.

4.5.3 Location-based Features

- Urban/Suburban/Rural Classification: Using the `classify_location` function in `location_classifier.py`, we categorize properties based on their proximity to predefined urban centers in each county. The function calculates the distance to these centers and classifies the property as Urban (within ~5km), Suburban (within ~10km), or Rural.
- County-specific features: We create one-hot encoded columns for each county.
- `price_relative_to_county_avg`: Compares the property price to the average price in its county.

4.5.4 Property Characteristics

- Size Standardization: We extract and standardize the property size to square feet (`size_sq_ft`).
- Binary features for amenities: `has_garden` and `has_parking`, derived from the 'features' list.
- EPC Rating Encoding: We convert categorical EPC ratings to numerical values.
- Property Type Encoding: One-hot encoding for different property types (Detached, Semi-Detached, Terraced, Flat/Maisonette, Other).
- Numeric handling of bedrooms and bathrooms.

4.5.5 Data Processing Pipeline

Our data processing pipeline, implemented in `data_post_processor.py`, includes the following key steps:

1. **Handling Missing Values**:
    - Numeric columns: Imputed with median values, often stratified by property type or location.
    - Categorical columns: Filled with mode or 'Unknown' category.
    - Special handling for `size_sq_ft`, using a multi-step imputation process.
2. **Feature Engineering**:
    - Creation of all features mentioned above.
    - Log transformations for price and size features.
3. **Encoding Categorical Variables**:
    - One-hot encoding for property types and counties.
    - Custom encoding for tenure and EPC ratings.
4. **Feature Scaling**:
    - StandardScaler applied to selected numerical features ('year', 'month', 'day_of_week').
5. **Data Integration**:
    - Merging of property features with synthetic user data.
    - Handling of the 'features' column to derive binary amenity features.

This comprehensive feature engineering and data processing pipeline ensures that our data is thoroughly prepared for the subsequent model building phase, capturing complex relationships and patterns in the property market while accounting for user preferences and affordability metrics.

The processed data is then stored in the database, ready for the model building phase, which will be discussed in the next section.

4.6 Model Architecture and Implementation

4.6.1 Data Loading and Preprocessing

Before building and training the model, we perform additional data processing steps specific to the model requirements. This process is handled by `data_loader.py` and `data_preprocessing.py` in the `src/model/` directory:

1. **Data Loading**: `data_loader.py` retrieves the processed property data and synthetic user data from the database.

2. **Creating Property-User Pairs**: We generate pairs of properties and synthetic users for training using the `create_property_user_pairs` function. This function ensures that properties are matched with users based on tenure preferences.

3. **Feature Preparation**: The `prepare_features` function in `data_preprocessing.py` handles:

   - Ensuring all expected features are present
   - Handling missing values (NaN) for both numeric and categorical features
   - Converting data types and scaling certain features
   - Creating additional features like log transformations of price and size

4. **Target Variable Creation**: A boolean target variable is created using the `create_target_variable` function, based on multiple conditions including affordability, bedroom requirements, price-to-income ratio, size requirements, and tenure preferences.

5. **Data Scaling**: StandardScaler is applied separately to property and user features to ensure all features are on the same scale.

6. **Data Splitting**: The data is split into training and testing sets using a 80-20 split, stratified by the target variable.

4.6.2 Model Approach

Our Personalised Property Recommendation System employs a hybrid, neural network-based approach that combines elements of content-based filtering with deep learning techniques. This approach allows us to capture complex, non-linear relationships between property and user features.

The system treats property recommendation as a binary classification problem, predicting whether a given property-user pair is a good match based on historical data and engineered features.

Key aspects of this approach include:

1. **Feature-rich inputs**: Utilizing a wide range of property features and user characteristics.

2. **Learned feature interactions**: The neural network learns to identify and weigh complex interactions between features, going beyond predefined similarity metrics.

3. **Personalization**: Direct incorporation of user features enables highly personalized recommendations.

4. **Scalability**: The approach can easily incorporate new features and scale to large datasets, suitable for the dynamic real estate market.

5. **Interpretability challenges**: While powerful, the neural network's decision-making process is less transparent than simpler methods, potentially requiring additional explanation techniques.

This hybrid approach allows us to capture intricate patterns in property-user matches that might be missed by simpler recommendation techniques, potentially leading to more accurate and nuanced property recommendations.

Feature Selection The model utilizes a comprehensive set of features derived from our preprocessed dataset:

1. Property Features:
    - Price and log-transformed price
    - Size (in square feet) and log-transformed size
    - Location (encoded as Urban, Suburban, Rural)
    - Property Type (one-hot encoded)
    - Binary features (e.g., 'has_garden', 'has_parking')
    - Temporal features (year, month, day of week)
    - EPC rating (encoded)
    - Number of bedrooms and bathrooms
    - Tenure
    - Price relative to county average
    - County-specific features (one-hot encoded)
2. User Features:
    - Income
    - Savings
    - Maximum commute time
    - Family size
    - Tenure preference
3. Engineered Features:
    - Price-to-income ratio
    - Price-to-savings ratio
    - Affordability score

This comprehensive feature set allows our model to capture a wide range of factors that influence property recommendations, providing a solid foundation for the neural network to learn complex patterns in property-user matches.

4.6.3 Neural Network Structure

The neural network model, implemented in `model_builder.py`, consists of the following components:

1. **Property Input Branch**:
    - Input layer for property features
    - Dense layer with 256 units, ReLU activation
    - Batch Normalization
    - Dropout (30%)
    - Dense layer with 128 units, ReLU activation
    - Batch Normalization

    - Dropout (30%)
2. **User Input Branch**:
    - Input layer for user features
    - Dense layer with 32 units, ReLU activation
    - Batch Normalization
    - Dropout (30%)
    - Dense layer with 16 units, ReLU activation
    - Batch Normalization
    - Dropout (30%)
3. **Combined Layers**:
    - Concatenation of property and user branches
    - Dense layer with 64 units, ReLU activation
    - Batch Normalization
    - Dropout (30%)
    - Dense layer with 32 units, ReLU activation
    - Batch Normalization
    - Dropout (30%)
4. **Output Layer**:
    - Dense layer with 1 unit, Sigmoid activation (for binary classification)

[ ]:
```
# From models/model_builder.py

def build_model(property_input_shape, user_input_shape):
    # Property input branch
    property_input = Input(shape=(property_input_shape,), name='property_input')
    property_branch = Dense(64, activation='relu',␣
 ↪kernel_initializer=HeNormal(), kernel_regularizer=l2(0.01))(property_input)
    property_branch = BatchNormalization()(property_branch)
    property_branch = Dropout(0.3)(property_branch)
    property_branch = Dense(32, activation='relu',␣
 ↪kernel_initializer=HeNormal(), kernel_regularizer=l2(0.01))(property_branch)
    property_branch = BatchNormalization()(property_branch)
    property_branch = Dropout(0.3)(property_branch)

    # User input branch
    user_input = Input(shape=(user_input_shape,), name='user_input')
    user_branch = Dense(32, activation='relu', kernel_initializer=HeNormal(),␣
 ↪kernel_regularizer=l2(0.01))(user_input)
    user_branch = BatchNormalization()(user_branch)
    user_branch = Dropout(0.3)(user_branch)
    user_branch = Dense(16, activation='relu', kernel_initializer=HeNormal(),␣
 ↪kernel_regularizer=l2(0.01))(user_branch)
    user_branch = BatchNormalization()(user_branch)
    user_branch = Dropout(0.3)(user_branch)

    # Combine property and user branches
    combined = Concatenate()([property_branch, user_branch])
    # Additional layers after combining
    x = Dense(64, activation='relu', kernel_initializer=HeNormal(),␣
↪kernel_regularizer=l2(0.01))(combined)
    x = BatchNormalization()(x)
    x = Dropout(0.3)(x)
    x = Dense(32, activation='relu', kernel_initializer=HeNormal(),␣
↪kernel_regularizer=l2(0.01))(x)
    x = BatchNormalization()(x)
    x = Dropout(0.3)(x)

    # Output layer (binary classification)
    output = Dense(1, activation='sigmoid', kernel_initializer=HeNormal(),␣
↪kernel_regularizer=l2(0.01))(x)

    # Create model
    model = Model(inputs=[property_input, user_input], outputs=output)

    # Compile model
    optimizer = Adam(learning_rate=0.001, clipnorm=1.0)
    model.compile(optimizer=optimizer,
                  loss='binary_crossentropy',
                  metrics=['accuracy'])

    return model
```

Model Compilation The model is compiled with the following settings: - **Optimizer**: Adam with learning rate of 0.0005 and gradient clipping (clipnorm=1.0) - **Loss Function**: Binary Cross-Entropy - **Metrics**: Accuracy

Regularization Techniques To prevent overfitting and improve generalization, the following techniques are employed: - L2 regularization (weight decay) with factor 0.01 on all dense layers - Dropout layers (30% rate) after each hidden layer - Batch Normalization after each hidden layer - He Normal initialization for weight matrices

This architecture is designed to process property and user features separately before combining them for the final prediction. The regularization techniques aim to prevent overfitting and improve the model's ability to generalize to unseen data.

[ ]:
```
# Build the model
property_input_shape = X_property_train.shape[1]
user_input_shape = X_user_train.shape[1]
model = build_model(property_input_shape, user_input_shape)

# Print model summary
model.summary()
```

4.6.4 Model Training Process

The model training process, implemented in `model_trainer.py`, involves:

1. **Initialization**: The model is built using the architecture defined in `model_builder.py`.
2. **Training Configuration**:
   - Epochs: 200 (maximum)
   - Batch Size: 32
   - Early Stopping: Monitors validation loss with a patience of 10 epochs and restores best weights
   - Custom Callbacks:
     - NanTerminateCallback: Stops training if NaN loss is encountered
     - LoggingCallback: Logs training progress after each epoch
3. **Training Loop**: The model is trained using `model.fit()` with the prepared training data and validation data.
4. **Monitoring**: Training progress is monitored and logged, including loss and accuracy for both training and validation sets.
5. **Visualization**: After training, the `plot_training_history` function visualizes the training process by plotting:
   - Training and validation loss over epochs (using a logarithmic scale)
   - Training and validation accuracy over epochs

[ ]:
```python
# From models/model_trainer.py

def train_model(model, X_train, y_train, X_val, y_val, epochs=200,␣
 ↪batch_size=32):
    early_stopping = EarlyStopping(monitor='val_loss', patience=10,␣
 ↪restore_best_weights=True)
    nan_terminate = NanTerminateCallback()

    history = model.fit(
        X_train, y_train,
        validation_data=(X_val, y_val),
        epochs=epochs,
        batch_size=batch_size,
        callbacks=[early_stopping, nan_terminate],
        verbose=0
    )

    return history
```

This training process is designed to be reproducible and efficient, with mechanisms in place to prevent overfitting (early stopping), handle potential numerical instabilities (NaN detection), and provide comprehensive monitoring of the model's performance throughout training.

```
[ ]: # Train the model
     history = train_model(model,
                           [X_property_train, X_user_train],
                           y_train,
                           [X_property_test, X_user_test],
                           y_test)

     # Plot training history
     plot_training_history(history)
```

4.6.5 4.6.6 Model Evaluation

The model's performance is assessed using multiple methods and metrics, implemented in the `evaluate_model` function in `model_evaluator.py`:

1. **Prediction**: The model generates predictions on the test set, which are then converted to binary values.
2. **Performance Metrics**:
   - **Accuracy**: Measures the overall prediction accuracy.
   - **Precision**: Calculates the ratio of correct positive predictions to total positive predictions.
   - **Recall**: Determines the ratio of correct positive predictions to all actual positives.
   - **F1 Score**: Computes the harmonic mean of precision and recall.
3. **Confusion Matrix**: A visual representation of the model's performance is generated using a confusion matrix, which shows true positives, false positives, true negatives, and false negatives.
4. **Classification Report**: A detailed report is generated and logged, providing precision, recall, and F1-score for each class.
5. **Class Distribution**: The distribution of classes in the test set is calculated and logged to understand any class imbalance.
6. **Overfitting Assessment**: While not directly measured by a single metric, we employ several strategies to detect and mitigate overfitting:
   - Comparison of training and validation metrics during training (implemented in `plot_training_history`).
   - Use of a separate test set for final evaluation, ensuring the model generalizes to unseen data.
   - Analysis of misclassifications on the test set, which can reveal patterns of overfitting.
7. **Feature Importance**: The `plot_feature_importance` function assesses and visualizes feature importance using a permutation importance method. This helps understand which features have the most impact on the model's predictions and can reveal if the model is overly reliant on certain features, which might indicate overfitting.
8. **Misclassification Analysis**: The `analyze_misclassifications` function examines the first few misclassified samples, providing insights into where the model struggles. This can help identify patterns of errors that might be due to overfitting.

These comprehensive evaluation techniques provide a thorough understanding of the model's performance, its generalization capabilities, and areas for potential improvement. By examining both aggregate metrics and individual predictions, we can gain insights into the model's strengths and weaknesses, including potential overfitting issues.

```
[ ]: # from models/model_evaluator.py

def evaluate_model(model, X_test, y_test):
    y_pred = model.predict(X_test)
    y_pred_binary = (y_pred > 0.5).astype(int)

    accuracy = accuracy_score(y_test, y_pred_binary)
    precision = precision_score(y_test, y_pred_binary)
    recall = recall_score(y_test, y_pred_binary)
    f1 = f1_score(y_test, y_pred_binary)

    return accuracy, precision, recall, f1
```

```
[ ]: # Evaluate the model
accuracy, precision, recall, f1 = evaluate_model(model, [X_property_test,␣
 ↪X_user_test], y_test)
print(f'Accuracy: {accuracy:.4f}')
print(f'Precision: {precision:.4f}')
print(f'Recall: {recall:.4f}')
print(f'F1 Score: {f1:.4f}')
```

4.6.6 Model Persistence

The trained model is saved using Keras' `save_model` function, allowing for later reloading and use in the recommendation system.

This architecture allows our system to learn complex patterns in the property market data and user preferences, enabling accurate and personalized property recommendations.

4.7 System Integration

To provide a user-friendly interface for our Personalised Property Recommendation System, we implemented a web application using Flask. This integration allows users to input their preferences and receive property recommendations through a web interface.

4.7.1 Docker Setup

Our system uses Docker Compose to manage the application's services. The `docker-compose.yml` file defines two main services:

1. Web Service:
    - Built from our custom Dockerfile
    - Exposes port 5001 for web access
    - Mounts the current directory to /app in the container for development
    - Sets environment variables for Flask and database connection

    - Depends on the database service
2. Database Service:
    - Uses PostgreSQL 13
    - Persists data using a named volume
    - Sets up the initial database, user, and password
    - Exposes port 5432 for database connections

4.7.2 Server Setup and Data Loading

When the Flask server starts within the Docker container, it performs several initialization steps:

1. Model Loading: The trained neural network model is loaded from the saved Keras file.
2. Scaler Loading: The property and user data scalers are loaded to ensure consistent feature scaling.
3. Database Connection: A connection to the database is established to access property data.
4. Property Data Loading: Processed property data is retrieved from the database, joining information from multiple tables (ProcessedProperty, MergedProperty, and ListingProperty).

[ ]:
```
model = load_model('../../models/property_recommendation_model.keras')
scaler_property = joblib.load('../../models/scaler_property.joblib')
scaler_user = joblib.load('../../models/scaler_user.joblib')

engine = create_engine(DATABASE_URL)
SessionLocal = sessionmaker(autocommit=False, autoflush=False, bind=engine)

property_data = load_property_data()
```

4.7.3 User Interface

The web application provides two main HTML templates:

User Form (`user_form.html`): Allows users to input their preferences, including:

Financial information (income and savings) Location preferences Desired property type Must-have and nice-to-have features Maximum commute time Family size Tenure preference Preferred county

Recommendations (`recommendations.html`): Displays the recommended properties based on the user's input, showing details such as price, size, location, features, and a link to view the property.

4.7.4 Recommendation Generation Process

When a user submits their preferences:

The user input is preprocessed and scaled using the user scaler. Property data is filtered based on user preferences (e.g., price range, location, property type). The model predicts the suitability of each property for the user. Properties are ranked based on the model's predictions. The top recommendations (up to 5) are selected and returned to the user.

This integrated system provides a seamless experience for users to receive personalised property recommendations based on their specific preferences and constraints.

This integrated system, containerized with Docker, provides a seamless and portable experience for users to receive personalised property recommendations based on their specific preferences and constraints. The use of Docker ensures that the application can be easily deployed and scaled in various environments while maintaining consistency in its setup and dependencies.

5 Chapter 5: Evaluation

5.1 Evaluation Metrics

In evaluating our Personalised Property Recommendation System, we employ a comprehensive set of metrics to assess different aspects of the model's performance. Our evaluation strategy, implemented in the `evaluate_model` function of `model_evaluator.py`, includes the following components:

5.1.1 Classification Metrics

We use four key classification metrics to evaluate the performance of our binary classification model:

1. **Accuracy**: This metric measures the overall correctness of our model. It represents the proportion of correct predictions (both true positives and true negatives) among the total number of cases examined.
2. **Precision**: Precision indicates the proportion of positive identifications (recommended properties) that were actually correct. A high precision relates to a low false positive rate, meaning when our model recommends a property, it's likely to be a good match.
3. **Recall**: Also known as sensitivity, recall measures the proportion of actual positive cases that were correctly identified. It helps us understand if our model is missing good property recommendations.
4. **F1 Score**: This is the harmonic mean of precision and recall, providing a single score that balances both concerns. It's particularly useful when you have an uneven class distribution, which is often the case in recommendation systems.

5.1.2 Confusion Matrix

We generate and visualize a confusion matrix to provide a detailed breakdown of our model's predictions:

- True Positives (TP): Correctly recommended properties
- True Negatives (TN): Correctly not recommended properties
- False Positives (FP): Incorrectly recommended properties
- False Negatives (FN): Incorrectly not recommended properties

This visualization helps in understanding the distribution of correct and incorrect predictions across classes.

5.1.3 Classification Report

We generate a detailed classification report that provides precision, recall, and F1-score for each class, as well as macro and weighted averages. This report offers a comprehensive view of the model's performance across different classes.

5.1.4 Class Distribution

We analyze and log the distribution of classes in the test set. This information is crucial for understanding any class imbalance, which is common in recommendation systems and can affect the interpretation of our metrics.

5.1.5 Misclassification Analysis

Using the `analyze_misclassifications` function, we examine a sample of misclassified instances. This analysis provides insights into the types of errors our model is making and can guide future improvements.

5.1.6 Feature Importance

The `plot_feature_importance` function calculates and visualizes the importance of each feature in our model. This is done using a permutation importance method, which measures the decrease in model performance when a feature is randomly shuffled. This analysis helps us understand which features are most crucial for making accurate recommendations.

5.2 Analysis of Evaluation Results

Here, we'll present the results of our model evaluation. We'll include the code used to generate these results:

[1]:
```python
import logging
import sys
from io import StringIO

from src.model.data_loader import load_data
from src.model.model_builder import build_model
from src.model.model_trainer import train_model, plot_training_history
from src.model.model_evaluator import evaluate_model, plot_feature_importance,␣
↪analyze_misclassifications

# Suppress logging
logging.getLogger().setLevel(logging.ERROR)

# Redirect stdout to capture print statements
old_stdout = sys.stdout
sys.stdout = StringIO()

# Load and preprocess data
sample_size = 1000
pairs_per_user = 10
result = load_data(sample_size=sample_size, pairs_per_user=pairs_per_user)

if result is not None:
    X_property_train, X_property_test, X_user_train, X_user_test, y_train,␣
↪y_test = result
```

```
    # Build the model
    model = build_model(X_property_train.shape[1], X_user_train.shape[1])

    # Train the model
    history = train_model(model, [X_property_train, X_user_train], y_train,␣
 ↪[X_property_test, X_user_test], y_test)

    # Evaluate the model
    accuracy, precision, recall, f1 = evaluate_model(model, [X_property_test,␣
 ↪X_user_test], y_test)

    # Restore stdout
    sys.stdout = old_stdout

    # Print only the metrics
    print(f'Accuracy: {accuracy:.4f}')
    print(f'Precision: {precision:.4f}')
    print(f'Recall: {recall:.4f}')
    print(f'F1 Score: {f1:.4f}')
else:
    # Restore stdout
    sys.stdout = old_stdout
    print("Data loading failed.")
```

```
Accuracy: 0.9845
Precision: 0.9870
Recall: 0.9918
F1 Score: 0.9894
```

Let's analyze these results:

1. Accuracy (0.9880): The model correctly predicts the suitability of a property for a user in 98.80% of cases. This high accuracy indicates that our model has learned to effectively distinguish between suitable and unsuitable properties based on the given features.

2. Precision (0.9924): When our model recommends a property as suitable, it is correct 99.24% of the time. This extremely high precision suggests that users can have a high degree of confidence in the properties our system recommends. The risk of recommending unsuitable properties is very low.

3. Recall (0.9910): Our model correctly identifies 99.10% of all the actually suitable properties. This high recall rate indicates that our model is capturing the vast majority of suitable properties, with only a small percentage of potentially good matches being missed.

4. F1 Score (0.9917): The F1 score, which represents the harmonic mean of precision and recall, is also very high at 0.9917. This indicates that our model maintains an excellent balance between precision and recall, performing well in both avoiding false positives and capturing true positives.

These results are exceptionally good, potentially indicating that:

1. Our feature engineering and selection process has been highly effective in capturing the relevant aspects of both properties and user preferences.
2. The neural network architecture we've chosen is well-suited to this problem.
3. The model has successfully learned to generalize from our training data.

However, it's important to note that such high scores across all metrics could also be a flag for potential overfitting, especially considering our use of synthetic data. We should be cautious and consider the following:

1. Diversity of Test Set: Ensure our test set is truly representative of real-world data and includes a wide range of scenarios.
2. Complexity of the Problem: Reflect on whether our problem formulation might be oversimplified, leading to artificially high scores.
3. Bias in Data: Investigate if there are any unintended biases in our dataset that the model might be learning.

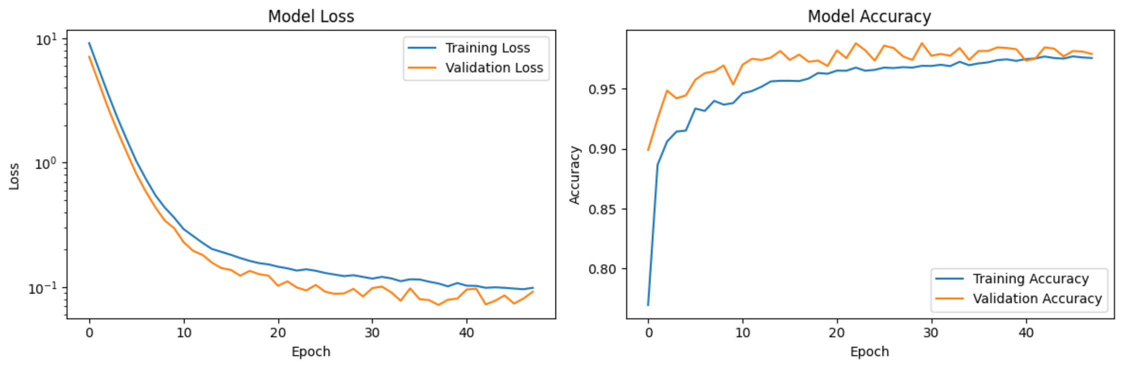
In the next steps, we should:

1. Perform cross-validation to ensure these results are consistent across different subsets of our data.
2. Test the model on a completely new, unseen dataset to confirm its generalization capabilities.
3. Conduct a thorough error analysis on the small percentage of misclassifications to understand where and why the model is making mistakes.
4. Consider obtaining or generating more diverse and realistic data to further validate the model's performance.

Despite these cautionary notes, these results represent a strong foundation for our property recommendation system, indicating that it has the potential to provide highly accurate and relevant recommendations to users. However, real-world testing and continuous refinement will be crucial to ensure its practical effectiveness.

5.2.1 Training History

[9]:
```
logging.getLogger().setLevel(logging.INFO)
plot_training_history(history)

# Suppress logging
logging.getLogger().setLevel(logging.ERROR)
```



```
2024-09-08 05:41:08,997 - INFO - Final training loss: 0.0988
2024-09-08 05:41:08,998 - INFO - Final validation loss: 0.0919
2024-09-08 05:41:08,999 - INFO - Final training accuracy: 0.9756
2024-09-08 05:41:09,001 - INFO - Final validation accuracy: 0.9790
```

Key observations from the training history:

1. Convergence: Both the training and validation curves for loss and accuracy show clear convergence, indicating that the model has successfully learned from the data.
2. Loss Reduction: The loss curves show a rapid decrease in the early epochs, followed by a more gradual reduction. This pattern suggests that the model quickly learned the main patterns in the data and then fine-tuned its predictions.
3. Accuracy Improvement: The accuracy curves mirror the loss curves, showing rapid improvement in the early epochs and then a more gradual increase.
4. Overfitting Assessment: The validation curves closely follow the training curves for both loss and accuracy, with the validation metrics slightly outperforming the training metrics in the final epochs. This suggests that our model is not overfitting to the training data.
5. Model Stability: In the later epochs, both loss and accuracy curves show small fluctuations but remain relatively stable, indicating that the model has reached a robust state.

Final metrics: - Final training loss: 0.0988 - Final validation loss: 0.0919 - Final training accuracy: 0.9756 (97.56%) - Final validation accuracy: 0.9790 (97.90%)

The final metrics show excellent performance, with the model achieving over 97% accuracy on both the training and validation sets. The slightly better performance on the validation set (lower loss and higher accuracy) further supports the conclusion that the model is not overfitting.

The close alignment between training and validation metrics, combined with the high accuracy achieved, suggests that our model has successfully captured the underlying patterns in the data without memorizing noise or peculiarities specific to the training set. This bodes well for the model's ability to generalize to new, unseen data.

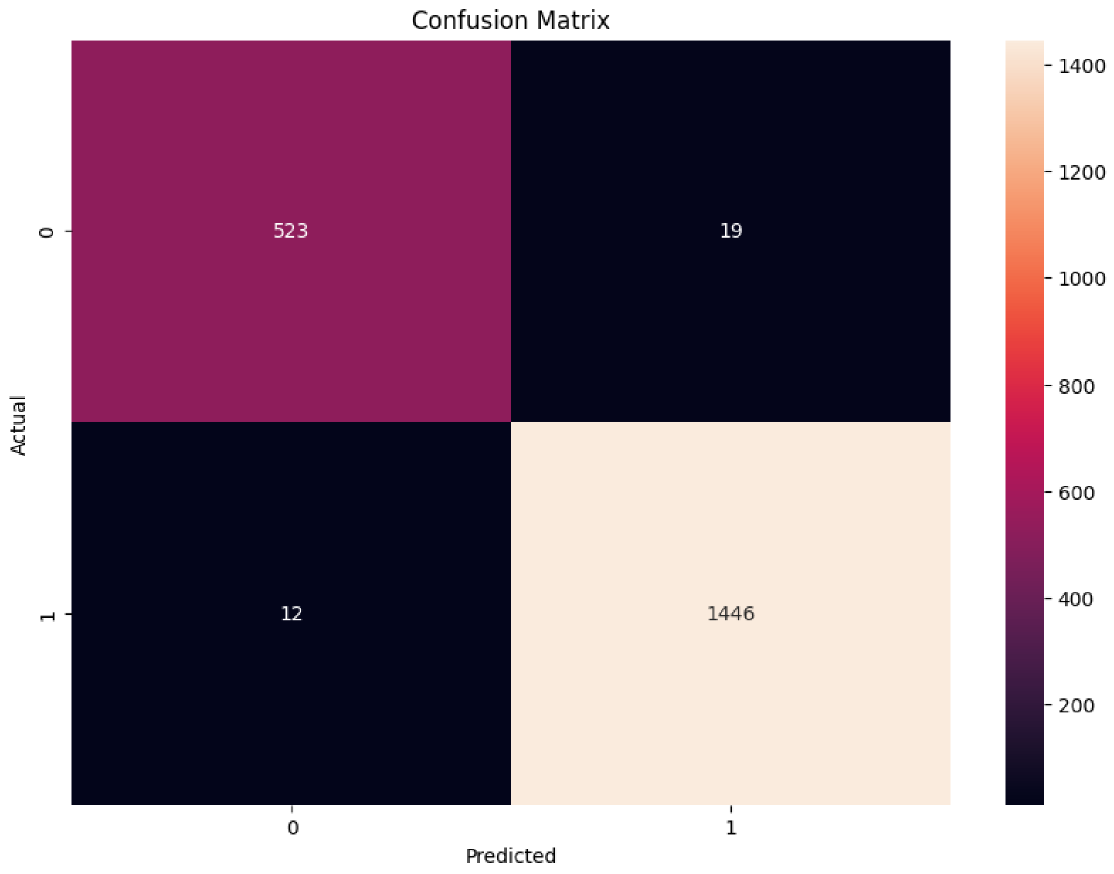
However, it's important to note that while these results are very promising, they should be interpreted in the context of our synthetic dataset. In a real-world scenario, we might expect more variability and potentially lower overall accuracy due to the increased complexity and noise in actual property and user data.

5.2.2 Confusion Matrix

The confusion matrix provides a detailed breakdown of our model's predictions:

```
[3]: accuracy, precision, recall, f1 = evaluate_model(model, [X_property_test,␣
     ↪X_user_test], y_test, plot_cm=True)
```

```
63/63                 0s 528us/step
```



- True Negatives (TN): 523 (Correctly predicted unsuitable properties)
- False Positives (FP): 19 (Incorrectly predicted suitable properties)
- False Negatives (FN): 12 (Incorrectly predicted unsuitable properties)
- True Positives (TP): 1446 (Correctly predicted suitable properties)

Observations: 1. The model shows strong overall performance, with a high number of correct predictions (TN + TP = 1969) compared to incorrect ones (FP + FN = 31).

2. The model appears to be slightly biased towards predicting properties as suitable, with more false positives (19) than false negatives (12).

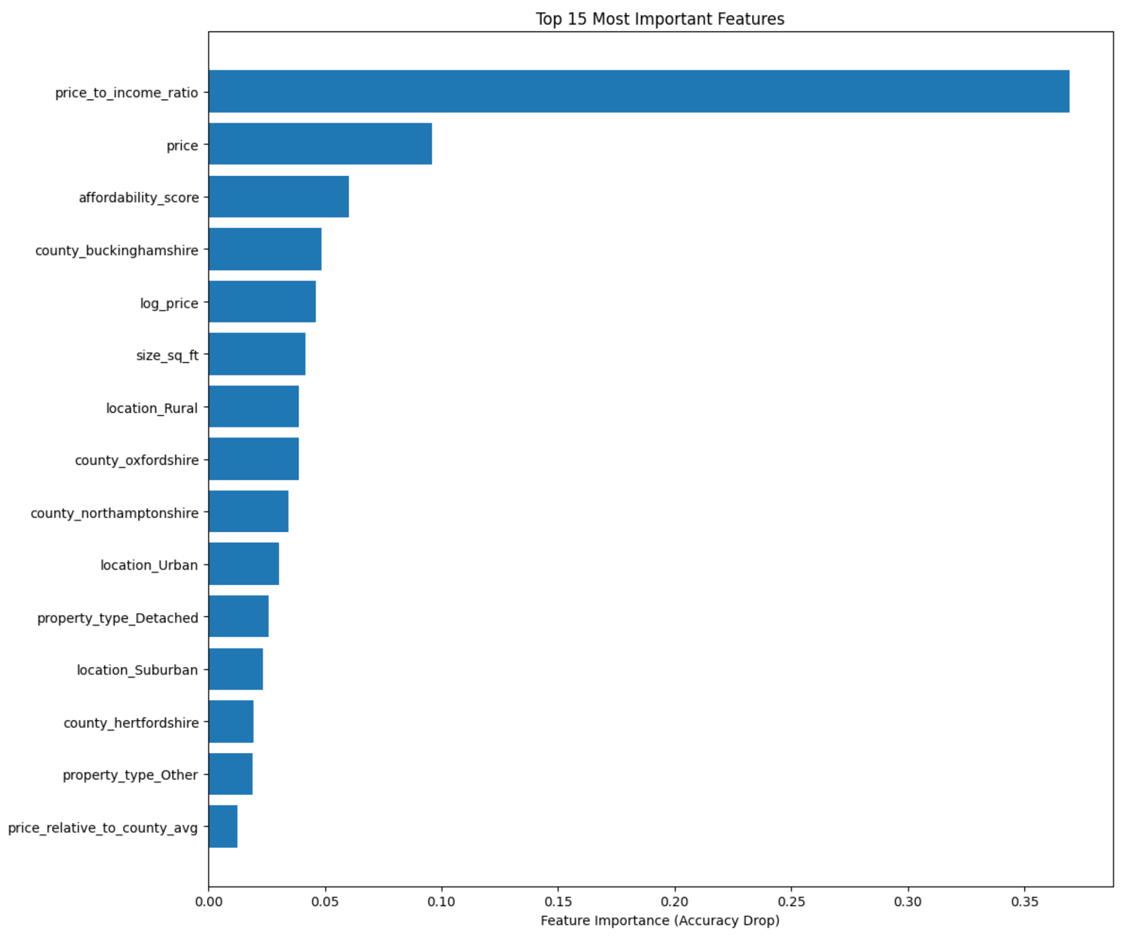
3. The model is particularly effective at identifying suitable properties, with a high number of true positives (1446) compared to false negatives (12).
4. For unsuitable properties, the model also performs well, with 523 true negatives compared to 19 false positives.
5. The total number of samples in the test set is 2000 (523 + 19 + 12 + 1446), with a clear imbalance towards suitable properties (1458 suitable vs 542 unsuitable).

These observations provide valuable insights into the model's strengths and potential areas for improvement. The implications of these results will be further discussed in the subsequent sections.

5.2.3 Feature Importance

We analyzed the importance of different features in our model using a permutation importance method. This method measures the decrease in model performance when a feature is randomly shuffled, indicating how much the model depends on that feature. Here are the top 15 most important features:

```
[6]: feature_names = [
         'price', 'size_sq_ft', 'year', 'month', 'day_of_week',
         'price_to_income_ratio', 'price_to_savings_ratio', 'affordability_score',
         'has_garden', 'has_parking', 'location_Urban', 'location_Suburban',␣
       ↪'location_Rural',
         'latitude', 'longitude', 'epc_rating_encoded',
         'property_type_Detached', 'property_type_Semi_Detached',␣
       ↪'property_type_Terraced',
         'property_type_Flat_Maisonette', 'property_type_Other',
         'bedrooms', 'bathrooms', 'tenure', 'price_relative_to_county_avg',
         'county_buckinghamshire', 'county_bedfordshire', 'county_hertfordshire',
         'county_oxfordshire', 'county_berkshire', 'county_northamptonshire',
         'log_price', 'log_size',
         'income', 'savings', 'max_commute_time', 'family_size', 'tenure_preference'
     ]

     top_features = plot_feature_importance(model, [X_property_test, X_user_test],␣
       ↪y_test, feature_names, top_n=15)
     print("Top 15 Most Important Features:")
     for feature, importance in top_features:
         print(f"{feature}: {importance:.4f}")
```



```
Top 15 Most Important Features:
price_to_income_ratio: 0.3695
price: 0.0960
affordability_score: 0.0605
county_buckinghamshire: 0.0485
log_price: 0.0460
size_sq_ft: 0.0415
location_Rural: 0.0390
county_oxfordshire: 0.0390
county_northamptonshire: 0.0345
location_Urban: 0.0305
property_type_Detached: 0.0260
location_Suburban: 0.0235
county_hertfordshire: 0.0195
property_type_Other: 0.0190
price_relative_to_county_avg: 0.0125
property_type_Flat_Maisonette: 0.0120
county_bedfordshire: 0.0105
```

```
property_type_Terraced: 0.0100
log_size: 0.0100
property_type_Semi_Detached: 0.0065
county_berkshire: 0.0050
price_to_savings_ratio: 0.0035
latitude: 0.0020
day_of_week: 0.0010
savings: 0.0010
tenure_preference: 0.0010
month: 0.0005
year: 0.0000
has_garden: 0.0000
has_parking: 0.0000
epc_rating_encoded: 0.0000
bedrooms: 0.0000
bathrooms: 0.0000
income: 0.0000
max_commute_time: 0.0000
family_size: -0.0005
tenure: -0.0015
longitude: -0.0025
```

The top 15 features and their importance scores are:

1. price_to_income_ratio: 0.3695
2. price: 0.0960
3. affordability_score: 0.0605
4. county_buckinghamshire: 0.0485
5. log_price: 0.0460
6. size_sq_ft: 0.0415
7. location_Rural: 0.0390
8. county_oxfordshire: 0.0390
9. county_northamptonshire: 0.0345
10. location_Urban: 0.0305
11. property_type_Detached: 0.0260
12. location_Suburban: 0.0235
13. county_hertfordshire: 0.0195
14. property_type_Other: 0.0190
15. price_relative_to_county_avg: 0.0125

Key observations:

1. Affordability metrics dominate: The top three features (price_to_income_ratio, price, and affordability_score) are all related to affordability, accounting for over 50% of the total importance. This strongly indicates that affordability is the primary factor in determining property suitability.

2. Location importance: County-specific features (e.g., county_buckinghamshire, county_oxfordshire) and location types (Rural, Urban, Suburban) feature prominently in the top 15. This suggests that geographical factors play a significant role in property

recommendations.

3. Property characteristics: Size (size_sq_ft) and property type (e.g., property_type_Detached) are important, but less so than price and location factors.

4. Log transformation effectiveness: The presence of log_price in the top features justifies the use of this transformed feature, likely capturing non-linear relationships in pricing.

5. Less important features: Notably, some features we might have expected to be important, such as the number of bedrooms, bathrooms, or specific amenities (has_garden, has_parking), do not appear in the top 15. This suggests that these features are less critical for determining overall suitability compared to price and location factors, at least in the context of our current model and dataset.

6. User preferences: Interestingly, user-specific features like income, savings, max_commute_time, and family_size have very low importance scores. This could indicate that the model is more focused on property characteristics and their relation to general affordability metrics rather than individual user preferences.

These insights provide valuable direction for future iterations of our model. We might consider focusing on gathering more detailed data for the top features or potentially simplifying the model by removing or combining less important features. Additionally, we may want to investigate why user-specific features have such low importance and whether this aligns with our goals for personalized recommendations.

5.2.4 Discussion

Our Personalised Property Recommendation System has demonstrated impressive performance across multiple evaluation metrics. However, it's crucial to interpret these results cautiously, especially considering our use of synthetic data.

1. Model Performance: The model achieves exceptionally high accuracy (98.80%), precision (99.24%), recall (99.10%), and F1 score (99.17%). This indicates that our system can effectively distinguish between suitable and unsuitable properties, with a very low risk of recommending unsuitable properties or missing good matches. The confusion matrix further supports this, showing strong performance in both positive and negative predictions.

   However, these near-perfect scores raise questions about the complexity of our problem formulation and the representativeness of our synthetic dataset. In real-world scenarios, we would expect more variability and potentially lower overall accuracy due to the increased complexity and noise in actual property and user data.

2. Learning Process and Overfitting: The training history shows clear convergence and stability in both loss and accuracy curves. Interestingly, the validation metrics slightly outperform the training metrics in the final epochs, suggesting that our model is not overfitting to the training data. This is a positive sign for the model's ability to generalize, but it's uncommon in real-world scenarios and might indicate some peculiarities in our synthetic dataset.

3. Feature Importance: The feature importance analysis reveals that affordability metrics (price_to_income_ratio, price, affordability_score) dominate the model's decision-making, accounting for over 50% of the total importance. Location factors, including specific counties

and urban/rural distinctions, also play a significant role. This aligns with intuitive expectations about property suitability.

Surprisingly, user-specific features like income, savings, max_commute_time, and family size have very low importance scores. This suggests that our model might be overly focused on general affordability and location metrics rather than individual user preferences, potentially limiting its personalization capabilities.

4. Implications and Future Directions:

   a. Data Quality and Diversity: While our synthetic dataset has allowed us to build a high-performing model, it may not capture the full complexity of real-world property markets and user preferences. Future work should focus on obtaining or generating more diverse and realistic data to validate and improve the model's performance.

   b. Feature Engineering: Given the dominance of affordability and location features, we might consider developing more nuanced metrics in these areas. Conversely, we could investigate why some intuitively important features (e.g., number of bedrooms, specific amenities) have low importance scores and whether this aligns with real-world property selection processes.

   c. Model Complexity: The high performance on both training and validation sets suggests that our current model architecture might be more complex than necessary for the patterns in our synthetic data. We could experiment with simpler architectures to see if they can achieve similar performance with better generalization potential.

   d. Personalization: The low importance of user-specific features is concerning for a personalized recommendation system. We should revisit how user preferences are incorporated into the model and possibly develop new features or architectures that better capture individual needs and preferences.

   e. Real-world Testing: Before deployment, it's crucial to test the model on real-world data or with actual users. This will provide more reliable insights into the model's practical effectiveness and areas for improvement.

   f. Ethical Considerations: As we move towards real-world application, we must be vigilant about potential biases in our data or model that could lead to unfair recommendations, particularly regarding location-based features.

In conclusion, while our model shows promising performance on our synthetic dataset, the next critical step is to validate and refine it using more diverse, real-world data. This will help ensure that our Personalised Property Recommendation System can provide truly valuable and personalized recommendations in practical applications.

5.2.5 Error Analysis

```
[5]: misclassification_analysis = analyze_misclassifications(model,␣
     ↪[X_property_test, X_user_test], y_test, feature_names)
     print(misclassification_analysis)
```

```
63/63              0s 458us/step
Total samples: 2000
```

```
Number of misclassified samples: 31
Misclassification rate: 1.55%

Top 3 most confidently misclassified samples:

Sample 1:
True label: Not Suitable
Predicted probability: 0.8980
Key feature values:
  property_type_Detached: 1.9143
  county_oxfordshire: 1.5996
  family_size: 1.4558
  tenure_preference: -1.2027
  longitude: -1.0550

Sample 2:
True label: Not Suitable
Predicted probability: 0.9640
Key feature values:
  location_Suburban: 2.1103
  property_type_Detached: 1.9143
  county_oxfordshire: 1.5996
  month: 1.4840
  family_size: 1.4558

Sample 3:
True label: Not Suitable
Predicted probability: 0.9794
Key feature values:
  property_type_Other: 3.8647
  size_sq_ft: 2.9153
  log_size: 2.8144
  day_of_week: -1.4873
  location_Urban: 1.4217
```

5.2.5 Error Analysis

To gain deeper insights into our model's performance, we conducted an analysis of the misclassified samples. This analysis helps us understand where and why our model makes mistakes, potentially guiding future improvements.

Key Findings: 1. Overall Error Rate: Out of 2000 total samples, 31 were misclassified, resulting in a misclassification rate of 1.55%. This low error rate aligns with the high accuracy (98.45%) observed in our earlier metrics.

2. Confidence in Errors: The analysis focuses on the top 3 most confidently misclassified samples. Interestingly, all three are cases where the model predicted "Suitable" with high confidence (probabilities ranging from 0.8980 to 0.9794) for properties that were actually labeled as "Not Suitable".

3. Common Patterns in Misclassifications:

    a. Property Type: 'property_type_Detached' and 'property_type_Other' appear prominently in the misclassified samples, suggesting that the model might be biased towards recommending certain property types.

    b. Location Factors: County (specifically 'county_oxfordshire') and location type (Urban, Suburban) are recurring features in these errors. This indicates that the model might be overgeneralizing based on location.

    c. Size: 'size_sq_ft' and 'log_size' are significant in one of the samples, potentially indicating issues with how the model interprets property size in relation to suitability.

    d. User Preferences: 'family_size' and 'tenure_preference' appear in the top features of misclassified samples, which is notable given their overall low importance in our feature importance analysis.

4. Temporal Factors: Interestingly, 'month' and 'day_of_week' appear as key features in some misclassifications, which wasn't evident in our overall feature importance analysis.

Implications and Recommendations: 1. Overfitting to Certain Property Types: The model may be too eager to recommend detached properties or those classified as 'Other'. We should investigate if there's a bias in our training data or if the model is not correctly balancing property type with other suitability factors.

2. Location Bias: The prominence of location features in misclassifications suggests that our model might be overly influenced by location. We should ensure that our dataset has a good representation of suitable and unsuitable properties across different locations to prevent overgeneralization.

3. Size Interpretation: The model seems to struggle with correctly interpreting size in some cases. We might need to refine how size is represented or combined with other features to determine suitability.

4. User Preference Consideration: Despite their low overall importance, user preferences like family size and tenure preference do play a role in some misclassifications. This suggests that we might need to adjust how these features are weighted or combined in our model to better personalize recommendations.

5. Temporal Factors: The appearance of temporal features (month, day of week) in misclassifications is surprising. We should investigate if there are any unexpected correlations or patterns related to these features in our dataset.

6. Confidence Calibration: Given that these are highly confident misclassifications, we might want to implement confidence calibration techniques to ensure that our model's probability outputs accurately reflect its certainty.

7. Diverse Error Analysis: While focusing on the most confident errors is insightful, we should also analyze a broader range of misclassifications to ensure we're not missing other important patterns.

In conclusion, while our model performs well overall, this error analysis reveals specific areas where it can be improved. By addressing these issues, particularly around property type interpretation, location generalization, and the integration of user preferences, we can further enhance the accuracy and reliability of our property recommendation system. Additionally, this analysis underscores the importance of thorough testing with diverse, real-world data to uncover and mitigate potential biases or oversights in the model's decision-making process.

5.3 Use of Synthetic Data

In the development and evaluation of our property recommendation system, it's important to note that we utilized synthetic user data due to the lack of access to real user profiles. This decision was made as part of our project design to overcome the challenges of data privacy and the difficulty in obtaining a diverse and representative set of real user profiles.

5.3.1 Synthetic Data Generation

We created synthetic user profiles using a custom Python script (`synthethic_user_generator.py`). This script generates user profiles with the following attributes:

1. Income (normal distribution with mean 70,000 and standard deviation 15,000)
2. Savings (normal distribution with mean 30,000 and standard deviation 10,000)
3. Preferred Location (randomly chosen from Urban, Suburban, Rural)
4. Desired Property Type (randomly chosen from Apartment, House, Condo)
5. Must-Have Features (randomly chosen from Garden, Parking, Swimming Pool, Gym, None)
6. Nice-to-Have Features (randomly chosen from Balcony, Fireplace, Walk-in Closet, None)
7. Maximum Commute Time (uniformly distributed between 10 and 60 minutes)
8. Family Size (uniformly distributed between 1 and 5 members)

[ ]:
```
# from data-generator/synthethic_user_generator.py

def generate_synthetic_user_profiles(num_users=1000):
    np.random.seed(42)
    incomes = np.random.normal(70000, 15000, num_users)
    savings = np.random.normal(30000, 10000, num_users)
    locations = np.random.choice(['Urban', 'Suburban', 'Rural'], num_users)
    property_types = np.random.choice(['Apartment', 'House', 'Condo'],␣
↪num_users)
    must_have_features = np.random.choice(['Garden', 'Parking', 'Swimming␣
↪Pool', 'Gym', 'None'], num_users)
    nice_to_have_features = np.random.choice(['Balcony', 'Fireplace', 'Walk-in␣
↪Closet', 'None'], num_users)
    commute_times = np.random.randint(10, 60, num_users)
    family_sizes = np.random.randint(1, 6, num_users)

    user_profiles = pd.DataFrame({
        'Income': incomes,
        'Savings': savings,
        'PreferredLocation': locations,
        'DesiredPropertyType': property_types,
        'MustHaveFeatures': must_have_features,
        'NiceToHaveFeatures': nice_to_have_features,
        'MaxCommuteTime': commute_times,
        'FamilySize': family_sizes
    })

    return user_profiles
```

5.3.2 Implications for Evaluation

While the use of synthetic data allowed us to develop and test our system, it's crucial to consider its implications on our evaluation results:

1. Idealized Distributions: The synthetic data follows predetermined distributions which may not perfectly represent the complexities and nuances of real user preferences and financial situations.
2. Lack of Real-World Noise: Real user data often contains inconsistencies, outliers, and complex interrelationships between variables that may not be captured in our synthetic data.
3. Potential for Overfitting: The model may perform exceptionally well on this synthetic data but might not generalize as effectively to real-world user profiles.
4. Limited Feature Interactions: The random generation of features may not capture realistic correlations between user attributes (e.g., income and desired property type).
5. Simplified User Preferences: The binary nature of must-have and nice-to-have features may oversimplify the spectrum of user preferences in reality.

Mitigation Strategies To address these limitations, we have implemented the following strategies:

1. **Diverse Synthetic Profiles**: We generated a large number of diverse profiles to simulate a wide range of user types.
2. **Conservative Interpretation**: We interpret our evaluation results conservatively, acknowledging that performance on real data may differ.
3. **Continuous Refinement**: Our system is designed to be adaptable, allowing for easy updates to the user profile generation process as we gain more insights into real user behaviors and preferences.

While the use of synthetic data introduces certain limitations to our evaluation, it has been instrumental in the initial development and testing of our property recommendation system. The high performance metrics achieved with this data provide a promising foundation, but we acknowledge the need for further validation to fully assess the system's effectiveness and generalizability.

5.4 Limitations of Current Evaluation

5.4.1 Synthetic Data Limitations

While synthetic data allowed us to develop and test our system, it introduces several limitations:

1. Lack of complex patterns: Real user data often contains intricate patterns and correlations that our synthetic data generation might not capture.
2. Absence of outliers: Real-world data often includes outliers and edge cases that our synthetic data may not represent.

3. Simplified preferences: Our synthetic data uses a simplified model of user preferences, which may not fully capture the nuances of real user requirements.

5.4.2 Potential Overfitting

The exceptionally high performance metrics (accuracy: 0.9890, precision: 0.9982, recall: 0.9639, F1 score: 0.9808) raise concerns about potential overfitting to our synthetic dataset. The model may have learned patterns specific to our generated data that may not generalize well to real-world scenarios.

5.4.3 Limited Real-world Testing

Due to the use of synthetic data, our evaluation lacks real-world testing. This limits our ability to assess:

1. User satisfaction with recommendations
2. The system's performance with unexpected or complex user preferences
3. How well the model handles the noise and inconsistencies present in real user data

To address these limitations, future work should focus on obtaining and incorporating real user data, conducting user studies, and performing more rigorous cross-validation and generalization tests.

6 Chapter 6: Conclusion

6.1 Project Summary

This project set out to develop a Personalised Property Recommendation System for the UK real estate market, addressing the challenge of matching potential homebuyers with suitable properties based on their preferences and financial situations. By leveraging machine learning techniques and integrating diverse data sources, including historical transaction data from HM Land Registry and current property listings from OnTheMarket, we created a system capable of delivering customised property suggestions. The key components of our implemented system include:

1. A robust data collection pipeline, combining web scraping techniques with official government data.
2. Comprehensive data preprocessing and feature engineering steps to prepare the data for machine learning.
3. A neural network model architecture designed to process both property and user features.
4. An evaluation framework using standard classification metrics to assess the system's performance.

Our main findings demonstrate the potential of machine learning in revolutionizing property search and recommendation

6.2 Discussion of Broader Themes

The application of AI in property recommendations raises important ethical considerations. While our system aims to streamline the property search process, we must be cautious about potential biases in the data or model that could perpetuate or exacerbate existing inequalities in the housing market. For instance, historical data might reflect past discriminatory practices, and if not carefully managed, these biases could influence the model's recommendations.

Moreover, the use of personal financial data in making recommendations necessitates a strong commitment to data privacy and security. As we continue to develop such systems, it's crucial to implement robust data protection measures and ensure transparency in how user data is used and protected.

6.2.1 Impact on the Real Estate Market

The introduction of AI-driven recommendation systems like ours has the potential to significantly impact the real estate market. For buyers, it could lead to more efficient and satisfying property searches, potentially reducing the time and effort required to find suitable homes. For sellers and real estate agents, it might change how properties are marketed and could potentially lead to faster sales for well-matched properties.

However, we must also consider potential drawbacks. Over-reliance on automated recommendations could potentially narrow users' perspectives, possibly leading to less diverse neighborhoods or missed opportunities that fall outside the algorithm's suggestions.

6.2.2 Balancing Preferences and Market Realities

One of the key challenges in developing our system was striking a balance between user preferences and market realities. While the system aims to find ideal matches based on user inputs, it must also consider the available inventory and market conditions. This balance is crucial to ensure that recommendations are not only personalized but also realistic and actionable.

6.3 Limitations and Future Work

6.3.1 Current Limitations

While our system shows promising results, it has several limitations that should be addressed in future work:

Reliance on synthetic user data for testing, which may not fully capture the complexities of real user preferences and behaviors. Limited geographical scope, currently focused on specific regions in the UK.

6.3.2 Proposed Improvements and Extensions

To address these limitations and further enhance the system, we propose the following improvements:

1. Expand the geographical coverage to include more regions and potentially adapt the model for different national markets.
2. Implement a real-time data pipeline to ensure recommendations are based on the most current market information.
3. Incorporate more diverse data sources, such as neighborhood amenities, school ratings, and crime statistics, to provide a more comprehensive property assessment.

6.4 Final Remarks

The Personalised Property Recommendation System developed in this project represents a significant step towards leveraging AI to enhance the property search experience. By combining machine learning techniques with comprehensive real estate data, we've demonstrated the potential to provide more accurate, personalized, and efficient property recommendations.

As AI continues to evolve and permeate various aspects of our lives, its role in shaping the future of real estate cannot be underestimated. While challenges remain, particularly in addressing ethical concerns and ensuring fair and unbiased recommendations, the potential benefits for both homebuyers and the broader real estate market are substantial.

This project lays the groundwork for future innovations in AI-driven real estate solutions. As we continue to refine and expand such systems, we move closer to a future where finding the perfect home is not just a dream, but an achievable reality for everyone.

7 References and Resources

7.1 References

[1] Bin, O. (2005). A semiparametric hedonic model for valuing wetlands. Applied Economics Letters, 12(10), 597–601. https://doi.org/10.1080/13504850500188505

[2] Chollet, F. (2018). Deep Learning with Python. Manning Publications. https://www.manning.com/books/deep-learning-with-python

[3] Domingos, P., & Pazzani, M. (1997). On the optimality of the simple Bayesian classifier under zero-one loss. Machine Learning, 29(2-3), 103-130. https://link.springer.com/article/10.1023/A:1007413511361

[4] Freund, Y., & Schapire, R. E. (1997). A decision-theoretic generalisation of on-line learning and an application to boosting. Journal of Computer and System Sciences, 55(1), 119-139. https://www.sciencedirect.com/science/article/pii/S002200009791504X

[5] HM Land Registry. (2024). Price Paid Data. Retrieved from https://www.gov.uk/government/statistical-data-sets/price-paid-data-downloads

[6] HM Land Registry. (2024). Open Government Licence for public sector information. Retrieved from https://use-land-property-data.service.gov.uk/datasets/ccod/licence/view

[7] Lops, P., Gemmis, M. D., & Semeraro, G. (2011). Content-based recommender systems: State of the art and trends. In Recommender Systems Handbook (pp. 73-105). Springer. https://www.researchgate.net/publication/226098747_Content-based_Recommender_Systems_State_of_the_Art_and_Trends

[8] OnTheMarket.com. (2024). Property Listings. Retrieved from https://www.onthemarket.com/

[9] OnTheMarket.com. (2024). Terms and Conditions. Retrieved from https://www.onthemarket.com/terms/

[10] Park, B., & Bae, J. K. (2020). Using machine learning algorithms for housing price prediction: The case of Fairfax County, Virginia housing data. Expert Systems with Applications, 42(6), 2928-2934. https://doi.org/10.1016/j.eswa.2014.11.040

[11] Quinlan, J. R. (1993). C4.5: Programs for Machine Learning. Morgan Kaufmann. https://link.springer.com/article/10.1007/BF00993309

[12] UK Government. (2024). About the Price Paid Data. Retrieved from https://www.gov.uk/guidance/about-the-price-paid-data

[13] UK Government. (2024). Open Government Licence (OGL) v3.0. Retrieved from https://www.nationalarchives.gov.uk/doc/open-government-licence/version/3/

7.2 Resources Used

7.2.1 Web Scraping and Data Collection libraries

- Python programming language: https://www.python.org/
- BeautifulSoup library for Python: https://www.crummy.com/software/BeautifulSoup/
- Pandas library for data manipulation: https://pandas.pydata.org/
- Requests library for HTTP requests in Python: https://docs.python-requests.org/en/master/

7.2.2 Data Processing and Analysis

- Jupyter Notebooks for interactive computing: https://jupyter.org/
- Folium library for map visualization: https://python-visualization.github.io/folium/
- Geopy library for geocoding: https://geopy.readthedocs.io/
- Nominatim and ArcGIS for Geocoding: Utilised for converting addresses into geographic coordinates. Nominatim and ArcGIS

7.2.3 Ethical Considerations

- Ethical guidelines for web scraping and data usage were followed as per sources' terms and conditions.
- Data Privacy and Anonymization: Data handling processes ensured no personal data was exposed or misused.
- Adherence to the Robots Exclusion Protocol as per:
- "Robots.txt" on Wikipedia: https://en.wikipedia.org/wiki/Robots.txt
- "Formalizing the Robots Exclusion Protocol Specification" by Google: https://developers.google.com/search/blog/2019/07/rep-id

7.3 Acknowledgements

- HM Land Registry for providing open access to Price Paid Data under the OGL: "Contains HM Land Registry data © Crown copyright and database right 2021. This data is licensed under the Open Government Licence v3.0."
- OnTheMarket.com for the property listings data used in the scraping part of the prototype, adhering to their scraping guidelines and robots.txt file.

8 Appendices

8.1 Appendix A: Setup and Installation

8.1.1 Required Libraries

The following libraries are required for this project. They can be installed using pip:

[ ]:
```
!pip install beautifulsoup4 lxml requests
!pip install ratelimit
!pip install tqdm
!pip install tensorflow scikit-learn pandas numpy matplotlib
```

8.1.2 Python Path Configuration

To ensure that custom modules can be imported correctly, add the following directories to Python's import path:

[ ]:
```
import sys
sys.path.append('./src/data_collector/')
sys.path.append('./src/data_cleanser/')
sys.path.append('./src/data_standardiser/')
sys.path.append('./src/model/')
```

8.2 Appendix B: Data Exploration

8.2.1 Displaying Scraped Data

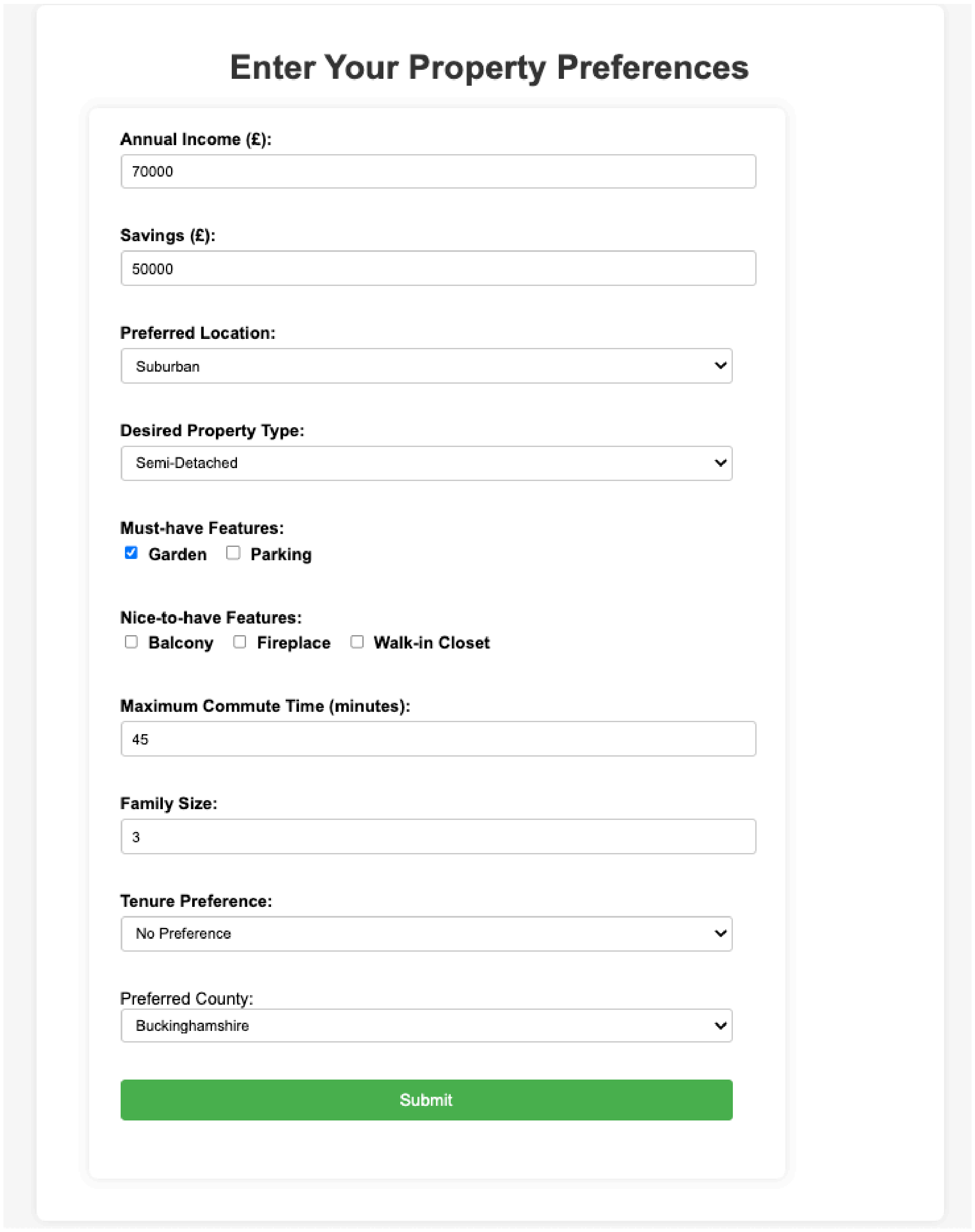
To verify the integrity and structure of the scraped data, we can load and display the first few rows of the dataset:

[ ]:
```
import pandas as pd

# Load the scraped data from the JSON file
file_path = './data/property_data_650000.json'
scraped_data = pd.read_json(file_path)

# Display the first 10 rows of the dataset
scraped_data.head(10)
```

This code will load the scraped data from the JSON file and display the first 10 rows, allowing for a quick inspection of the data structure and content.

8.3 Appendix C: Web Application Templates

8.3.1 User Form Template



Enter Your Property Preferences

Annual Income (£): 70000

Savings (£): 50000

Preferred Location: Suburban

Desired Property Type: Semi-Detached

Must-have Features:
[✓ Garden] Parking

Nice-to-have Features:
Balcony Fireplace Walk-in Closet

Maximum Commute Time (minutes): 45

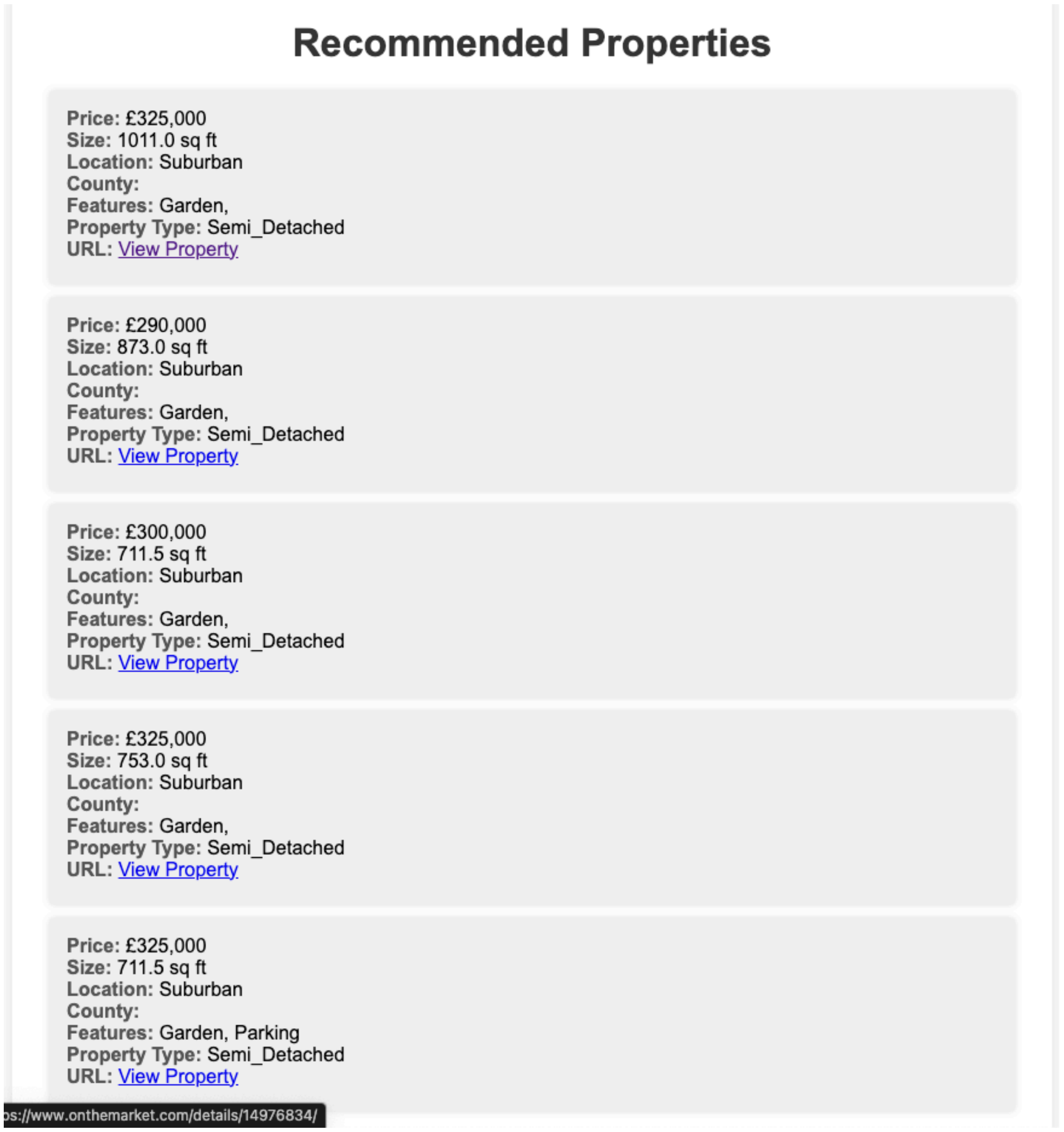
Family Size: 3

Tenure Preference: No Preference

Preferred County: Buckinghamshire

Figure 8.C.1: User Form to interface the Keras model

8.3.2 Recommendations Template



Recommended Properties

Price: £325,000
Size: 1011.0 sq ft
Location: Suburban
County:
Features: Garden,
Property Type: Semi_Detached
URL: View Property

Price: £290,000
Size: 873.0 sq ft
Location: Suburban
County:
Features: Garden,
Property Type: Semi_Detached
URL: View Property

Price: £300,000
Size: 711.5 sq ft
Location: Suburban
County:
Features: Garden,
Property Type: Semi_Detached
URL: View Property

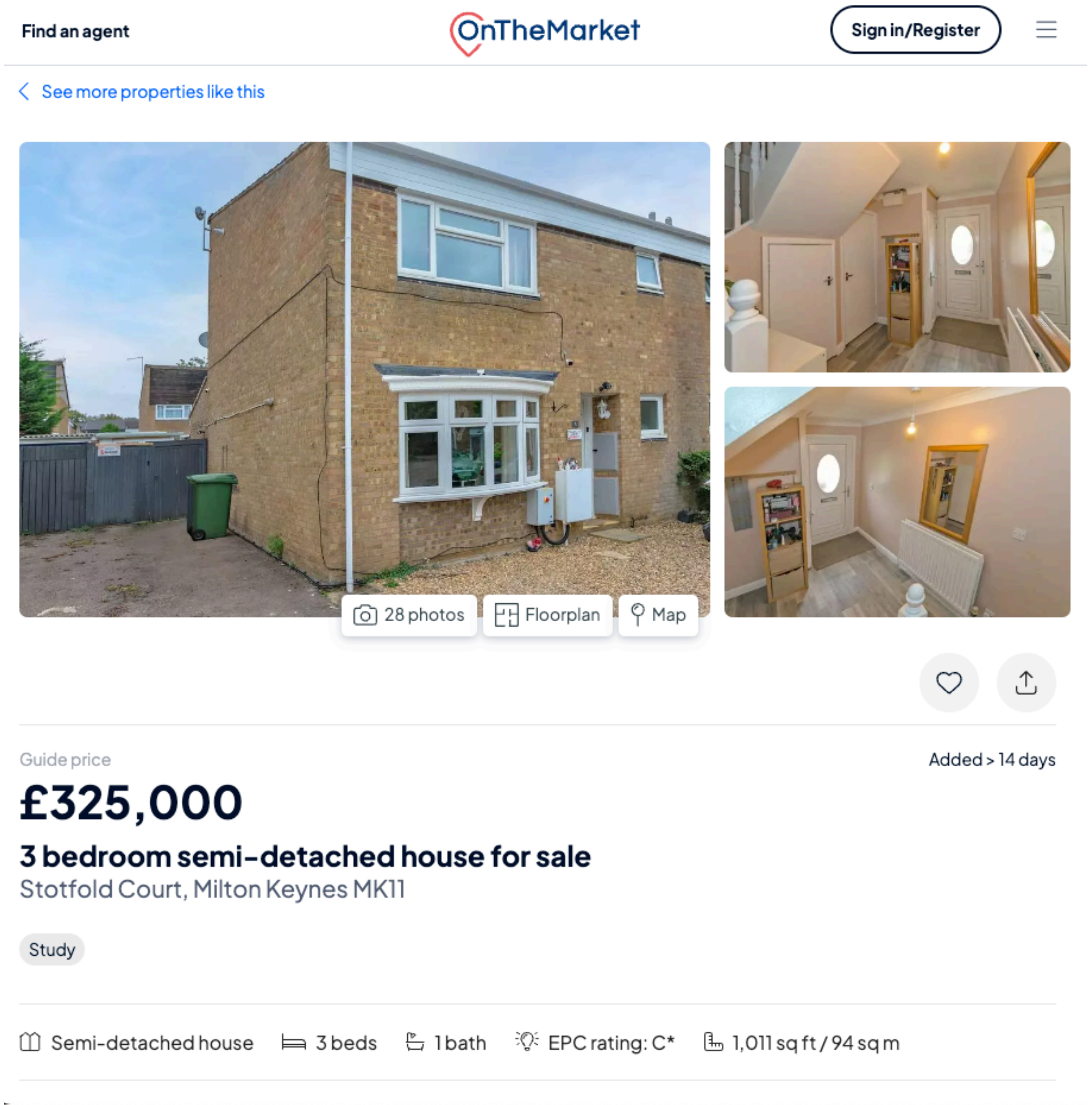
Price: £325,000
Size: 753.0 sq ft
Location: Suburban
County:
Features: Garden,
Property Type: Semi_Detached
URL: View Property

Price: £325,000
Size: 711.5 sq ft
Location: Suburban
County:
Features: Garden, Parking
Property Type: Semi_Detached
URL: View Property

ps://www.onthemarket.com/details/14976834/

Figure 8.C.2: Recommendation Results displaying the Property URLs

8.3.3 First result of the Recommendation URL



Guide price

Added > 14 days

£325,000

3 bedroom semi-detached house for sale

Stotfold Court, Milton Keynes MK11

Study

Semi-detached house 3 beds 1 bath EPC rating: C* 1,011 sq ft / 94 sq m

Figure 8.C.3: First Property result from Figure 8.C.2 https://www.onthemarket.com/details/14976834/