draft-report

July 22, 2024

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

1.1 1. Introduction

1.1.1 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.

1.1.2 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.

1.1.3 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.

1.1.4 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.

1.1.5 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.

1.1.6 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 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 2.2 Housing Price Prediction Models

2.2.1 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 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 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 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 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

2.6 2.6 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]: 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

[3]: Freund, Y., & Schapire, R. E. (1997). A decision-theoretic generalization 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

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

[5]: 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

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

3 Chapter 3: A Design

3.1 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 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 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 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 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 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 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 4.1 Data Collection and Preprocessing

4.1.1 Data Sources The Personalised Property Recommendation System relies on two primary data sources:

1. **HM Land Registry:** This dataset provides comprehensive historical transaction data for properties in Buckinghamshire, including sale prices, property types, and locations.
2. **OnTheMarket.com:** Current property listings were obtained through web scraping, providing up-to-date information on available properties.

4.1.2 Web Scraping Methodology To complement the historical data, we implemented a web scraping solution to collect current property listings from OnTheMarket.com. The scraping process is modularized into four main components:

1. **robot_check.py**: Ensures compliance with the website's scraping policies by interpreting the robots.txt file.
2. **crawler.py**: Discovers and collects relevant property listing URLs using requests and BeautifulSoup. It implements rate limiting to avoid overloading the server.
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.

Ethical considerations were prioritized, including respecting robots.txt, using a unique user agent, implementing rate limiting, and ensuring non-disruptive interaction with the website.

4.1.3 Data Cleaning and Standardization The data cleaning and standardization process is handled by two key scripts:

1. `data_cleanser_service.py`:
   - Assigns appropriate headers to the HM Land Registry dataset.
   - Loads and structures the data into a pandas DataFrame.
   - Filters the data to retain only properties in Buckinghamshire.
   - Converts dates and performs initial data cleaning.
2. `data_standardiser_service.py`:
   - Implements geocoding using Nominatim and ArcGIS to add latitude and longitude data.
   - Standardizes price formats across both datasets.
   - Normalizes addresses for consistency.
   - Merges the processed scraped data with the registry data.
   - Classifies property types consistently across the dataset.

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

# Add the directory containing custom modules 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/')
```

Modularity and Code Structure Our web scraping is structured into four main modules, each with a specific role. This modular approach ensures a clean separation of concerns, making the code more maintainable and scalable.

1. **`robot_check.py` - Respecting Site Policies**

Ensures compliance with the website's scraping policies by parsing and interpreting the robots.txt file using urllib.robotparser. This module is used by both crawler.py and data_collector_service.py.

[ ]:
```
# src/data-collector/robot_check.py
import urllib.robotparser


class RobotCheck:
    def __init__(self, robots_txt_url):
        self.parser = urllib.robotparser.RobotFileParser()
        self.parser.set_url(robots_txt_url)
        self.parser.read()

    def is_allowed(self, url, user_agent):
        return self.parser.can_fetch(user_agent, url)
```

```
    def get_crawl_delay(self, user_agent):
        delay = self.parser.crawl_delay(user_agent)
        return delay if delay is not None else 1
```

2. **crawler.py - Discovering URLs**

Crawls the target website, gathering relevant page URLs using `requests` and `BeautifulSoup`. Rate limiting is implemented with `ratelimit` to avoid overloading the server. This module is used by `data_collector_service.py`.

[ ]:
```
# src/data-collector/crawler.py
import time
import requests
from bs4 import BeautifulSoup
from robot_check import RobotCheck
from ratelimit import limits, sleep_and_retry


# 10 requests per minute
REQUESTS_PER_MINUTE = 10


@sleep_and_retry
@limits(calls=REQUESTS_PER_MINUTE, period=60)
def make_request(url, headers):
    return requests.get(url, headers=headers)


def get_property_urls(base_url, search_url, user_agent):
    headers = {'User-Agent': user_agent}
    robot_check = RobotCheck("https://www.onthemarket.com/robots.txt")
    property_urls = set()
    page_number = 1

    while search_url:
        if robot_check.is_allowed(search_url, user_agent):
            response = make_request(search_url, headers=headers)

            if response.status_code == 200:
                soup = BeautifulSoup(response.content, 'html.parser')
                links = soup.select('div.otm-PropertyCardMedia > div > a')
                current_page_urls = set()

                for link in links:
                    href = link.get('href')
                    if href and href.startswith('/details/'):  # Filtering␣
 ↪other links
                        full_url = 'https://www.onthemarket.com' + href
```

```
                        property_urls.add(full_url)
                        current_page_urls.add(full_url)

                # print(f"{len(current_page_urls)} URLs captured on Page␣
↪{page_number}.")

                # Find the next page URL
                next_page = soup.select_one('a[title="Next page"]')
                if next_page:
                    search_url = base_url + next_page['href']
                    page_number += 1
                else:
                    search_url = None

            else:
                print(f"Failed to retrieve the webpage: Status Code {response.
↪status_code}")
                break

            # Sleep for the crawl delay
            time.sleep(robot_check.get_crawl_delay(user_agent))
        else:
            print("Scraping is disallowed by the website's robots.txt for this␣
↪URL.")
            break

    return list(property_urls)
```

3. **scraper.py - Extracting Data**

Extracts specific data from web pages using `requests` and `BeautifulSoup`. This module is used by `data_collector_service.py`.

[ ]:
```
# src/data-collector/scraper.py
import requests
from bs4 import BeautifulSoup


def scrape_property_details(property_url, headers, counter):
    try:
        response = requests.get(property_url, headers=headers)
        if response.status_code == 200:
            soup = BeautifulSoup(response.content, 'html.parser')

            # Extracting data
            # Extracting the title
```

```
            title = soup.find('h1', class_='h4 md:text-xl leading-normal').
↪get_text(strip=True) if soup.find('h1', class_='h4 md:text-xl␣
↪leading-normal') else 'Title not available'

            # Extracting the address
            address_div = soup.find('div', class_='text-slate h4 font-normal␣
↪leading-none font-heading')
            address = address_div.get_text(strip=True) if address_div else␣
↪'Address not available'

            # Extracting the price
            price_div = soup.find('div', class_='otm-Price')
            if price_div:
                price = price_div.find('span', class_='price').
↪get_text(strip=True) if price_div.find('span', class_='price') else 'Price␣
↪not available'

                # Extracting the pricing qualifier if present
                qualifier_div = price_div.find('small', class_='qualifier')
                price_qualifier = qualifier_div.get_text(strip=True) if␣
↪qualifier_div else 'Price qualifier not available'
            else:
                price = 'Price not available'
                price_qualifier = 'Price qualifier not available'

            # Extracting the listing time
            listing_time_div = soup.find('div', class_='text-denim')
            if listing_time_div:
                listing_time = listing_time_div.find('small' ,␣
↪class_='font-heading').get_text(strip=True) if listing_time_div.find(
                    'small') else 'Listing time not available'
            else:
                listing_time = 'Listing time not available'

            # Extracting the property type
            property_type_div = soup.find('div', class_='otm-PropertyIcon')
            property_type = property_type_div.get_text(
                strip=True) if property_type_div else 'Property type not␣
↪available'

            details_div = soup.find('div', class_='font-heading text-xs flex␣
↪flex-wrap border-t border-b mb-6 md:text-md py-3 md:py-4 md:mb-9')
            if details_div:
                # Extracting the number of bedrooms
                bedrooms = 'Bedrooms info not available'
                for div in details_div.find_all('div'):
```

```
                    if 'bed' in div.get_text(strip=True).lower():
                        bedrooms = div.get_text(strip=True)
                        break

                # Extracting the number of bathrooms
                bathrooms = 'Bathrooms info not available'
                for div in details_div.find_all('div'):
                    if 'bath' in div.get_text(strip=True).lower():
                        bathrooms = div.get_text(strip=True)
                        break

                # Extracting the EPC rating
                epc_rating = 'EPC rating not available'
                for div in details_div.find_all('div'):
                    if 'epc rating' in div.get_text(strip=True).lower():
                        epc_rating = div.get_text(strip=True)
                        break

                # Extracting the property size
                size = 'Size info not available'
                for div in details_div.find_all('div'):
                    text = div.get_text(strip=True).lower()
                    if 'sq ft' in text or 'sq m' in text:
                        size = div.get_text(strip=True)
                        break

            # Extracting features
            features_section = soup.find('section', class_='otm-FeaturesList')
            features = []
            if features_section:
                feature_items = features_section.find_all('li',␣
↪class_='otm-ListItemOtmBullet')
                for item in feature_items:
                    feature_text = item.get_text(strip=True)
                    features.append(feature_text)

            return {
                'id': counter,
                'property_url': property_url,
                'title': title,
                'address': address,
                'price': price,
                'pricing_qualifier': price_qualifier,
                'listing_time': listing_time,
                'property_type': property_type,
                'bedrooms': bedrooms,
                'bathrooms': bathrooms,
```

```
                'epc_rating': epc_rating,
                'size': size,
                'features': features
            }
        else:
            print(f"Failed to retrieve the property page: Status Code {response.
↪status_code}")
            return {}
    except requests.exceptions.RequestException as e:
        return {'error': f"Request failed: {e}"}
    except Exception as e:
        return {'error': f"An unexpected error occurred: {e}"}
```

4. **`data_collector_service.py` - Orchestrating the Scraping Process**

Orchestrates the entire web scraping process. It calls on `crawler.py` to get URLs and then uses `scraper.py` to extract data from them. Handles errors, saves the data (usually in JSON), and can be run from the command line or Jupyter Notebook with adjustments.

[ ]:
```python
# src/data-collector/data_colletor_service.py
import json
from crawler import get_property_urls
from scraper import scrape_property_details
from robot_check import RobotCheck
from tqdm.notebook import tqdm
import time
import sys
import os

class DataCollectorService:
    def __init__(self, base_url, user_agent, max_price=120000):
        self.base_url = base_url
        self.user_agent = user_agent
        self.max_price = max_price
        self.robot_check = RobotCheck(f"{base_url}/robots.txt")

    @staticmethod
    def generate_price_segments(max_price, segment_size=100000):
        segments = []
        current_min = 0
        while current_min < max_price:
            current_max = min(current_min + segment_size, max_price)
            segments.append((current_min, current_max))
            current_min = current_max + 1
        return segments

    def collect_data_segment(self, min_price, max_price):
```

```
        search_url = f"{self.base_url}/for-sale/property/buckinghamshire/?
↪auction=false&min-price={min_price}&max-price={max_price}&new-home-flag=F&prop-types=bungal
        return get_property_urls(self.base_url, search_url, self.user_agent)

    def collect_data(self):
        price_segments = self.generate_price_segments(self.max_price)
        all_property_urls = set()
        all_property_data = []
        counter = 1

        for min_price, max_price in price_segments:
            segment_urls = self.collect_data_segment(min_price, max_price)
            all_property_urls.update(segment_urls)
            print(f"Segment {min_price}-{max_price}: Found {len(segment_urls)}␣
↪URLs.")

        print(f"Found a total of {len(all_property_urls)} property URLs from␣
↪all segments.")

        for url in tqdm(all_property_urls, desc="Scraping properties"):
            if self.robot_check.is_allowed(url, self.user_agent):
                headers = {'User-Agent': self.user_agent}
                data = scrape_property_details(url, headers, counter)
                if 'error' in data:
                    print(data['error'])
                else:
                    all_property_data.append(data)
                    counter += 1
                    # Respect the crawl delay
                    time.sleep(self.robot_check.get_crawl_delay(self.
↪user_agent))
            else:
                tqdm.write(f"Skipping {url}, disallowed by robots.txt.")

        self.save_data(all_property_data)

    def save_data(self, data):
        data_directory = './data'
        filename = f'./data/property_data_{self.max_price}.json'
        backup_filename = f'./data/property_data_{self.max_price}-backup.json'

        try:
            # Ensure data directory exists
            if not os.path.exists(data_directory):
                os.makedirs(data_directory)
                print(f"Created directory {data_directory}")
```

```
            # Rename existing files if they exist
            if os.path.exists(backup_filename):
                os.remove(backup_filename)
            if os.path.exists(filename):
                os.rename(filename, backup_filename)
                print(f"Renamed existing file to {backup_filename}")

            # Save the new data
            with open(filename, 'w') as file:
                json.dump(data, file, indent=4)
            print(f"Data saved successfully in {filename}")
        except IOError as e:
            print(f"An IOError occurred while saving data: {e}")
        except Exception as e:
            print(f"An unexpected error occurred while saving data: {e}")

if __name__ == "__main__":
    max_price_arg = 120000
    base_url = 'https://www.onthemarket.com'
    search_url = base_url + '/for-sale/property/buckinghamshire/?
 ↪auction=false&max-price=500000&min-price=10000&new-home-flag=F&prop-types=bungalows&prop-ty
    user_agent = 'StudentDataProjectScraper/1.0 (Contact: gor5@student.london.
 ↪ac.uk)'
    service = DataCollectorService(base_url, user_agent, max_price_arg)
    service.collect_data()
```

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
```

Displaying Scraped Data Verifying the integrity and structure of the scraped data is essential to ensure it aligns with our requirements and is ready for analysis.

[ ]:
```
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 5 to 10 rows of the dataset
scraped_data.head(10)
```

4.2 4.2 Data Analysis and Feature Engineering

4.2.1 4.2.1 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)
```

4.2.2 4.2.2 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)
```

4.2.4 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)
```

4.3 4.3 Feature Engineering

Feature engineering was performed to enhance the model's ability to capture relevant patterns:

1. Temporal Features: Extracted year, month, and day of the week from transaction dates.

2. Affordability Metrics: Created features such as price-to-income ratio and affordability score.
3. Location Encoding: One-hot encoded location types (Urban, Suburban, Rural).
4. Property Type Encoding: One-hot encoded property types.
5. Size Standardization: Converted various size measurements to a standard square footage metric.

4.3.1 Data Processing Pipeline

4.3.1.1 Data Post-Processing Before feeding data into our model, we perform several crucial post-processing steps using the `data_post_processor.py` script. This script prepares our raw data for machine learning tasks:

1. **Handling Missing Values**: We impute missing values in numeric columns with the median and in categorical columns with 'Unknown'.
2. **Feature Engineering**: We create new features such as:
   - Temporal features (year, month, day of week) from the transaction date
   - Affordability features (price-to-income ratio, price-to-savings ratio, affordability score)
   - Extracting numerical size from textual descriptions
   - Encoding EPC ratings
3. **Encoding Categorical Variables**: We perform one-hot encoding for property types and location types.
4. *Feature Scaling**: We use StandardScaler to scale selected numerical features.
5. **Feature Selection**: We select a final set of features for our machine learning model, including both original and engineered features.

This post-processing ensures our data is in the optimal format for our machine learning model.

4.3.1.2 Data Loading and Preprocessing In our Jupyter notebook, we use the following code to load and preprocess our data:

[16]:
```
# based on data_loader.py

import sys
sys.path.append('./src')

from model.data_loader import load_data
from model.data_loader import inspect_data
from model.model_builder import build_model
from model.model_trainer import train_model, plot_training_history
from model.model_evaluator import evaluate_model
import logging
import numpy as np

# Configure logging
logging.basicConfig(level=logging.INFO, format='%(asctime)s - %(levelname)s -␣
↪%(message)s')

# Set file paths and parameters
```

```
property_file_path = 'data/ml-ready-data/ml_ready_data.csv'
user_file_path = 'data/synthetic_user_profiles/synthetic_user_profiles.csv'
sample_size = 1000  # can be adjusted
pairs_per_user = 10  # can be adjusted

try:
    logging.info("Starting data loading process")
    X_property_train, X_property_test, X_user_train, X_user_test, y_train,␣
 ↪y_test = load_data(property_file_path, user_file_path, sample_size,␣
 ↪pairs_per_user)

    # logging.info("\nTraining set shapes:")
    # logging.info(f"X_property_train: {X_property_train.shape}")
    # logging.info(f"X_user_train: {X_user_train.shape}")
    # logging.info(f"y_train: {y_train.shape}")

    # logging.info("\nTest set shapes:")
    # logging.info(f"X_property_test: {X_property_test.shape}")
    # logging.info(f"X_user_test: {X_user_test.shape}")
    # logging.info(f"y_test: {y_test.shape}")

    # Additional data exploration
    # logging.info(f"\nTarget distribution in training set:")
    # logging.info(f"Positive samples: {sum(y_train)}")
    # logging.info(f"Negative samples: {len(y_train) - sum(y_train)}")

    # Inspect the data
    # logging.info("\nInspecting training data:")
    # inspect_data(X_property_train, X_user_train, y_train)
    # logging.info("\nInspecting test data:")
    # inspect_data(X_property_test, X_user_test, y_test)

except Exception as e:
    logging.error("An error occurred during data loading:", exc_info=True)
```

```
2024-07-22 13:05:37,935 - INFO - Starting data loading process
2024-07-22 13:05:37,937 - INFO - Loading property data from: data/ml-ready-
data/ml_ready_data.csv
2024-07-22 13:05:37,943 - INFO - Property data shape: (1000, 18)
2024-07-22 13:05:37,943 - INFO - Loading user data from:
data/synthetic_user_profiles/synthetic_user_profiles.csv
2024-07-22 13:05:37,946 - INFO - User data shape: (1000, 8)
2024-07-22 13:05:37,946 - INFO - Creating property-user pairs
2024-07-22 13:05:38,476 - INFO - Pairs created. Shape: (10000, 26)
2024-07-22 13:05:38,477 - WARNING - Location features were missing. Added dummy
columns.
2024-07-22 13:05:38,479 - INFO - Handling NaN values
2024-07-22 13:05:38,483 - INFO - Calculating affordability features
```

```
2024-07-22 13:05:38,485 - INFO - Creating target variable
2024-07-22 13:05:38,485 - INFO - Data preprocessing completed in 0.55 seconds
```

This code demonstrates how we use our `data_loader.py` module to load and preprocess the data, creating property-user pairs and splitting the data into training and testing sets.

4.4 4.4 Model Architecture and Implementation

4.4.1 4.4.1 Content-Based Filtering Approach

Our Personalised Property Recommendation System employs a content-based filtering approach, which recommends items based on their features and the user's preferences. This method is particularly suitable for our real estate context as it can effectively match property characteristics with user requirements.

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

1. Property Features:
   - Price
   - Size (in square feet)
   - Location (encoded as Urban, Suburban, Rural)
   - Property Type (e.g., apartment, detached house)
   - Additional binary features like 'has_garden', 'has_parking'
   - Year, month, and day of week (from transaction date)
2. User Features:
   - Income
   - Savings
   - Maximum commute time
   - Family size
3. Engineered Features:
   - Price-to-income ratio
   - Price-to-savings ratio
   - Affordability score

4.4.2 4.4.2 Neural Network Architecture

We implemented a neural network using TensorFlow and Keras to create our recommendation model. The architecture is designed to process both property and user features simultaneously, allowing for personalized recommendations.

[ ]:
```
# 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
```

Key features of this architecture:

1. **Dual Input Branches**: Separate branches for property and user features allow the model to learn specific patterns in each domain before combining them.

2. **Dense Layers**: Multiple dense layers with ReLU activation capture complex relationships in the data.
3. **Regularization**: L2 regularization and dropout layers help prevent overfitting.
4. **Batch Normalization**: Improves model stability and performance.
5. **Output Layer**: A single neuron with sigmoid activation for binary classification (suitable/not suitable property).

[17]:
```
# 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()
```

```
Model: "functional_5"
```

Layer (type)	Output Shape	Param #	Connected to
property_input (InputLayer)	(None, 17)	0	-
user_input (InputLayer)	(None, 4)	0	-
dense_14 (Dense)	(None, 64)	1,152	property_input[0…
dense_16 (Dense)	(None, 32)	160	user_input[0][0]
batch_normalizatio… (BatchNormalizatio…	(None, 64)	256	dense_14[0][0]
batch_normalizatio… (BatchNormalizatio…	(None, 32)	128	dense_16[0][0]
dropout_12 (Dropout)	(None, 64)	0	batch_normalizat…
dropout_14 (Dropout)	(None, 32)	0	batch_normalizat…
dense_15 (Dense)	(None, 32)	2,080	dropout_12[0][0]
dense_17 (Dense)	(None, 16)	528	dropout_14[0][0]
batch_normalizatio… (BatchNormalizatio…	(None, 32)	128	dense_15[0][0]

batch_normalizatio… (BatchNormalizatio…	(None, 16)	64	dense_17[0][0]
dropout_13 (Dropout)	(None, 32)	0	batch_normalizat…
dropout_15 (Dropout)	(None, 16)	0	batch_normalizat…
concatenate_2 (Concatenate)	(None, 48)	0	dropout_13[0][0], dropout_15[0][0]
dense_18 (Dense)	(None, 64)	3,136	concatenate_2[0]…
batch_normalizatio… (BatchNormalizatio…	(None, 64)	256	dense_18[0][0]
dropout_16 (Dropout)	(None, 64)	0	batch_normalizat…
dense_19 (Dense)	(None, 32)	2,080	dropout_16[0][0]
batch_normalizatio… (BatchNormalizatio…	(None, 32)	128	dense_19[0][0]
dropout_17 (Dropout)	(None, 32)	0	batch_normalizat…
dense_20 (Dense)	(None, 1)	33	dropout_17[0][0]

Total params: 10,129 (39.57 KB)

Trainable params: 9,649 (37.69 KB)

Non-trainable params: 480 (1.88 KB)

4.4.3 4.4.3 Model Training Process

The training process is managed by the model_trainer.py script, which includes the following key components:

1. **Data Splitting**: The dataset is split into training and validation sets using an 80-20 ratio.
2. **Batch Size and Epochs**: The model is trained with a batch size of 32 for up to 200 epochs.

3. **Early Stopping**: To prevent overfitting, an early stopping callback monitors validation loss with a patience of 10 epochs.
4. **Learning Rate**: An initial learning rate of 0.001 is used with the Adam optimizer.

[ ]:
```
# 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 and handle potential numerical instabilities.

[18]:
```
# 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)
```

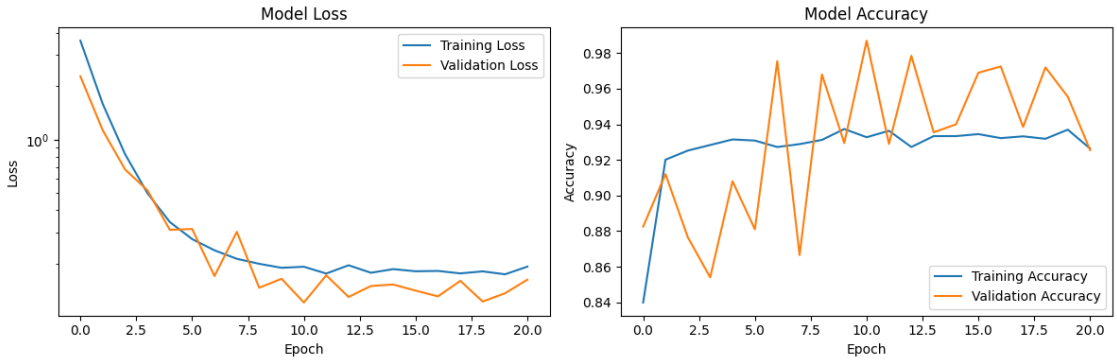
```
Epoch 1/200
250/250             2s 2ms/step -
accuracy: 0.7458 - loss: 4.5000 - val_accuracy: 0.8825 - val_loss: 2.2701
Epoch 2/200
250/250             0s 934us/step -
accuracy: 0.9176 - loss: 1.8778 - val_accuracy: 0.9120 - val_loss: 1.1287
Epoch 3/200
250/250             0s 895us/step -
accuracy: 0.9239 - loss: 0.9598 - val_accuracy: 0.8765 - val_loss: 0.6790
Epoch 4/200
250/250             0s 911us/step -
accuracy: 0.9313 - loss: 0.5494 - val_accuracy: 0.8540 - val_loss: 0.5187
```

```
Epoch 5/200
250/250             0s 900us/step -
accuracy: 0.9334 - loss: 0.3702 - val_accuracy: 0.9080 - val_loss: 0.3107
Epoch 6/200
250/250             0s 934us/step -
accuracy: 0.9332 - loss: 0.2864 - val_accuracy: 0.8810 - val_loss: 0.3141
Epoch 7/200
250/250             0s 948us/step -
accuracy: 0.9349 - loss: 0.2308 - val_accuracy: 0.9755 - val_loss: 0.1708
Epoch 8/200
250/250             0s 960us/step -
accuracy: 0.9308 - loss: 0.2127 - val_accuracy: 0.8665 - val_loss: 0.3028
Epoch 9/200
250/250             0s 970us/step -
accuracy: 0.9371 - loss: 0.1924 - val_accuracy: 0.9680 - val_loss: 0.1467
Epoch 10/200
250/250             0s 971us/step -
accuracy: 0.9371 - loss: 0.1936 - val_accuracy: 0.9295 - val_loss: 0.1648
Epoch 11/200
250/250             0s 948us/step -
accuracy: 0.9326 - loss: 0.1933 - val_accuracy: 0.9870 - val_loss: 0.1212
Epoch 12/200
250/250             0s 946us/step -
accuracy: 0.9339 - loss: 0.1762 - val_accuracy: 0.9290 - val_loss: 0.1724
Epoch 13/200
250/250             0s 964us/step -
accuracy: 0.9279 - loss: 0.1937 - val_accuracy: 0.9785 - val_loss: 0.1302
Epoch 14/200
250/250             0s 950us/step -
accuracy: 0.9345 - loss: 0.1762 - val_accuracy: 0.9355 - val_loss: 0.1500
Epoch 15/200
250/250             0s 949us/step -
accuracy: 0.9332 - loss: 0.1934 - val_accuracy: 0.9400 - val_loss: 0.1529
Epoch 16/200
250/250             0s 953us/step -
accuracy: 0.9377 - loss: 0.1733 - val_accuracy: 0.9690 - val_loss: 0.1413
Epoch 17/200
250/250             0s 962us/step -
accuracy: 0.9302 - loss: 0.1862 - val_accuracy: 0.9725 - val_loss: 0.1312
Epoch 18/200
250/250             0s 958us/step -
accuracy: 0.9283 - loss: 0.1855 - val_accuracy: 0.9385 - val_loss: 0.1605
Epoch 19/200
250/250             0s 956us/step -
accuracy: 0.9353 - loss: 0.1771 - val_accuracy: 0.9720 - val_loss: 0.1227
Epoch 20/200
250/250             0s 954us/step -
accuracy: 0.9380 - loss: 0.1685 - val_accuracy: 0.9555 - val_loss: 0.1366
```

```
Epoch 21/200
250/250             0s 936us/step -
accuracy: 0.9304 - loss: 0.1905 - val_accuracy: 0.9255 - val_loss: 0.1629

2024-07-22 13:05:57,082 - INFO - Epoch 1/21
2024-07-22 13:05:57,082 - INFO - loss: 3.6046 - accuracy: 0.8399 - val_loss:
2.2701 - val_accuracy: 0.8825
2024-07-22 13:05:57,083 - INFO - Epoch 2/21
2024-07-22 13:05:57,083 - INFO - loss: 1.5880 - accuracy: 0.9201 - val_loss:
1.1287 - val_accuracy: 0.9120
2024-07-22 13:05:57,083 - INFO - Epoch 3/21
2024-07-22 13:05:57,083 - INFO - loss: 0.8279 - accuracy: 0.9252 - val_loss:
0.6790 - val_accuracy: 0.8765
2024-07-22 13:05:57,084 - INFO - Epoch 4/21
2024-07-22 13:05:57,084 - INFO - loss: 0.4997 - accuracy: 0.9284 - val_loss:
0.5187 - val_accuracy: 0.8540
2024-07-22 13:05:57,084 - INFO - Epoch 5/21
2024-07-22 13:05:57,084 - INFO - loss: 0.3439 - accuracy: 0.9315 - val_loss:
0.3107 - val_accuracy: 0.9080
2024-07-22 13:05:57,084 - INFO - Epoch 6/21
2024-07-22 13:05:57,085 - INFO - loss: 0.2756 - accuracy: 0.9309 - val_loss:
0.3141 - val_accuracy: 0.8810
2024-07-22 13:05:57,085 - INFO - Epoch 7/21
2024-07-22 13:05:57,085 - INFO - loss: 0.2383 - accuracy: 0.9273 - val_loss:
0.1708 - val_accuracy: 0.9755
2024-07-22 13:05:57,085 - INFO - Epoch 8/21
2024-07-22 13:05:57,086 - INFO - loss: 0.2135 - accuracy: 0.9289 - val_loss:
0.3028 - val_accuracy: 0.8665
2024-07-22 13:05:57,086 - INFO - Epoch 9/21
2024-07-22 13:05:57,086 - INFO - loss: 0.2000 - accuracy: 0.9312 - val_loss:
0.1467 - val_accuracy: 0.9680
2024-07-22 13:05:57,086 - INFO - Epoch 10/21
2024-07-22 13:05:57,086 - INFO - loss: 0.1900 - accuracy: 0.9374 - val_loss:
0.1648 - val_accuracy: 0.9295
2024-07-22 13:05:57,087 - INFO - Epoch 11/21
2024-07-22 13:05:57,087 - INFO - loss: 0.1926 - accuracy: 0.9327 - val_loss:
0.1212 - val_accuracy: 0.9870
2024-07-22 13:05:57,087 - INFO - Epoch 12/21
2024-07-22 13:05:57,087 - INFO - loss: 0.1764 - accuracy: 0.9364 - val_loss:
0.1724 - val_accuracy: 0.9290
2024-07-22 13:05:57,087 - INFO - Epoch 13/21
2024-07-22 13:05:57,088 - INFO - loss: 0.1963 - accuracy: 0.9273 - val_loss:
0.1302 - val_accuracy: 0.9785
2024-07-22 13:05:57,088 - INFO - Epoch 14/21
2024-07-22 13:05:57,088 - INFO - loss: 0.1781 - accuracy: 0.9334 - val_loss:
0.1500 - val_accuracy: 0.9355
2024-07-22 13:05:57,088 - INFO - Epoch 15/21
2024-07-22 13:05:57,088 - INFO - loss: 0.1868 - accuracy: 0.9334 - val_loss:
```

```
0.1529 - val_accuracy: 0.9400
2024-07-22 13:05:57,088 - INFO - Epoch 16/21
2024-07-22 13:05:57,089 - INFO - loss: 0.1816 - accuracy: 0.9345 - val_loss:
0.1413 - val_accuracy: 0.9690
2024-07-22 13:05:57,089 - INFO - Epoch 17/21
2024-07-22 13:05:57,089 - INFO - loss: 0.1824 - accuracy: 0.9323 - val_loss:
0.1312 - val_accuracy: 0.9725
2024-07-22 13:05:57,089 - INFO - Epoch 18/21
2024-07-22 13:05:57,090 - INFO - loss: 0.1767 - accuracy: 0.9333 - val_loss:
0.1605 - val_accuracy: 0.9385
2024-07-22 13:05:57,090 - INFO - Epoch 19/21
2024-07-22 13:05:57,090 - INFO - loss: 0.1814 - accuracy: 0.9319 - val_loss:
0.1227 - val_accuracy: 0.9720
2024-07-22 13:05:57,090 - INFO - Epoch 20/21
2024-07-22 13:05:57,090 - INFO - loss: 0.1746 - accuracy: 0.9370 - val_loss:
0.1366 - val_accuracy: 0.9555
2024-07-22 13:05:57,091 - INFO - Epoch 21/21
2024-07-22 13:05:57,091 - INFO - loss: 0.1928 - accuracy: 0.9264 - val_loss:
0.1629 - val_accuracy: 0.9255
```




```
2024-07-22 13:05:57,446 - INFO - Final training loss: 0.1928
2024-07-22 13:05:57,446 - INFO - Final validation loss: 0.1629
2024-07-22 13:05:57,447 - INFO - Final training accuracy: 0.9264
2024-07-22 13:05:57,447 - INFO - Final validation accuracy: 0.9255
```

4.4.4 4.4.4 Model Evaluation

After training, the model's performance is assessed using multiple metrics::

1. **Accuracy**: Evaluates the overall accuracy of the model's predictions..
2. **Precision**: Refers to the ratio of accurate positive identifications to the total number of positive identifications.
3. **Recall**: Refers to the ratio of accurately identified positive instances to the total number of genuine positive cases.
4. **F1 Score**: The harmonic mean of precision and recall. It is used to provide a fair estimate

of the model's performance.

These metrics are calculated using the `evaluate_model` function in `model_evaluator.py`:

[ ]:
```
# 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
```

[19]:
```
# 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}')
```

```
63/63            0s 1ms/step
Accuracy: 0.9870
Precision: 0.9861
Recall: 0.9692
F1 Score: 0.9776
```

4.5 4.5 System Integration

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

4.5.1 4.5.1 Model and Data Loading:

The trained machine learning model and necessary data are loaded when the server starts, ensuring efficient response times for user requests.

[ ]:
```
## based on src/webserver/app.py

model = load_model('../../models/property_recommendation_model.h5')
scaler_property = joblib.load('../../models/scaler_property.joblib')
scaler_user = joblib.load('../../models/scaler_user.joblib')
property_data = load_property_data()
```

5 Chapter 5: Evaluation

5.1 5.1 Evaluation Metrics

In evaluating our property recommendation system, we use four key metrics: accuracy, precision, recall, and F1 score are metrics used to evaluate the performance of classification models. Each of these indicators offers distinct perspectives on the performance of our 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.
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.

These metrics are crucial for our property recommendation system as they help us understand not just how often our model is correct (accuracy), but also how well it's performing in terms of recommending suitable properties (precision) and not missing out on potentially good matches (recall).

5.2 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:

[20]:
```
# 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}')
```

```
63/63             0s 506us/step
Accuracy: 0.9870
Precision: 0.9861
Recall: 0.9692
F1 Score: 0.9776
```

Let's analyze these results:

1. **Accuracy (0.9870)**: Our model achieves an impressive 98.70% accuracy, indicating that it correctly predicts the suitability of a property for a user in the vast majority of cases. This high accuracy suggests that our model has learned to effectively distinguish between suitable and unsuitable properties based on the given features.
2. **Precision (0.9861)**: With a precision of 99.61%, when our model recommends a property as suitable, it is correct 98.61% of the time. This extremely high precision indicates 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 (0.9692)**: Our model correctly identifies 96.92% of all the actually suitable properties. While slightly lower than our precision, this is still a very good recall rate. It suggests 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 (0.9776)**: The F1 score of 0.9776 represents a strong balance between precision and recall. This high F1 score indicates that our model is 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. 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.

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.

5.3 5.3 Analysis of Evaluation Results

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 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)

[ ]:
```python
# 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 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.

5.3.3 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 5.4 Limitations of Current Evaluation

5.4.1 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 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 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 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 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 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 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 6.3 Limitations and Future Work

6.3.1 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 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 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 7. References and Resources

7.1 7.1 References

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

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

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

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

[5]: HM Land Registry Price Paid Data License. "Open Government Licence for public sector information." Available at: https://use-land-property-data.service.gov.uk/datasets/ccod/licence/view.

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

7.2 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 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.