

Rental Recommendation System Using CRISP-DM CMPE-255 Project Report

Brandon Jacklyn
FNU Sameer
Sarvesh Borkar
Shruti Goyal

Abstract

We present a rental recommendation web service that uses a robust clustering model to enhance the property search experience. Finding the perfect rental property is often a time-consuming and overwhelming process for users, given the vast array of listings available online. Following the CRISP-DM methodology this report goes through the steps of data collection, data cleansing, and data preprocessing. Then after performing feature engineering and reducing dimensionality, the CRISP-DM steps of modeling and evaluation explore the various clustering models we created. We trained 6 different clustering models and after comparing the silhouette, davies bouldin, and calinski harabasz scores, we found that HDBSCAN performed the best. Finally, we describe how the model is deployed to our web application.

Introduction

Users frequently struggle to locate properties that align with their specific preferences and to explore comparable alternatives. To address this issue, we have developed a rental recommendation web service that leverages a clustering-based recommendation engine. This application allows users to search for rental listings in a chosen area and view recommendations for similar listings when exploring a specific property. Our clustering model, trained on scraped property data, identifies patterns and groups similar listings based on features like location, price, size, and amenities. This approach streamlines the rental search process, offering users relevant and personalized recommendations, making it faster and more user-friendly.

In the first step of CRISP-DM, *Business Understanding*, we surveyed the market to see what the current state-of-art is. While existing platforms like Craigslist, Zillow, and Realtor.com offer basic filters for rental searches, they either don't have property recommendations, or their recommendations exist to push paid advertisements. Our application allows users to search for rental listings in a desired area and view personalized recommendations for similar properties when exploring a specific listing.

The next step of CRISP-DM is *Data Understanding* which includes sourcing the data for our project. Data is downloaded through a web scraper pipeline running on a nightly cron, which collects and processes rental listings from platforms like Zillow and Realtor.com using Kafka and Apache Flink. This ensures the database remains comprehensive and up-to-date.

We performed extensive EDA in order to gain insights on the data we collected, and find any data quality issues that need to be fixed in the *Data Preparation* phase. EDA consisted of over 15 custom charts/plots, as well as automated EDA using Sweetviz. Using these insights we cleaned and prepared the data by removing duplicate rows from duplicate scraping of the same property, dropping irrelevant columns, converting types, removing outliers, and performing imputation. Feature engineering created new columns through binning, one-hot encoding, label encoding. Numeric columns were scaled and standardized, and dimensionality using PCA, UMAP, and Autoencoders were all explored.

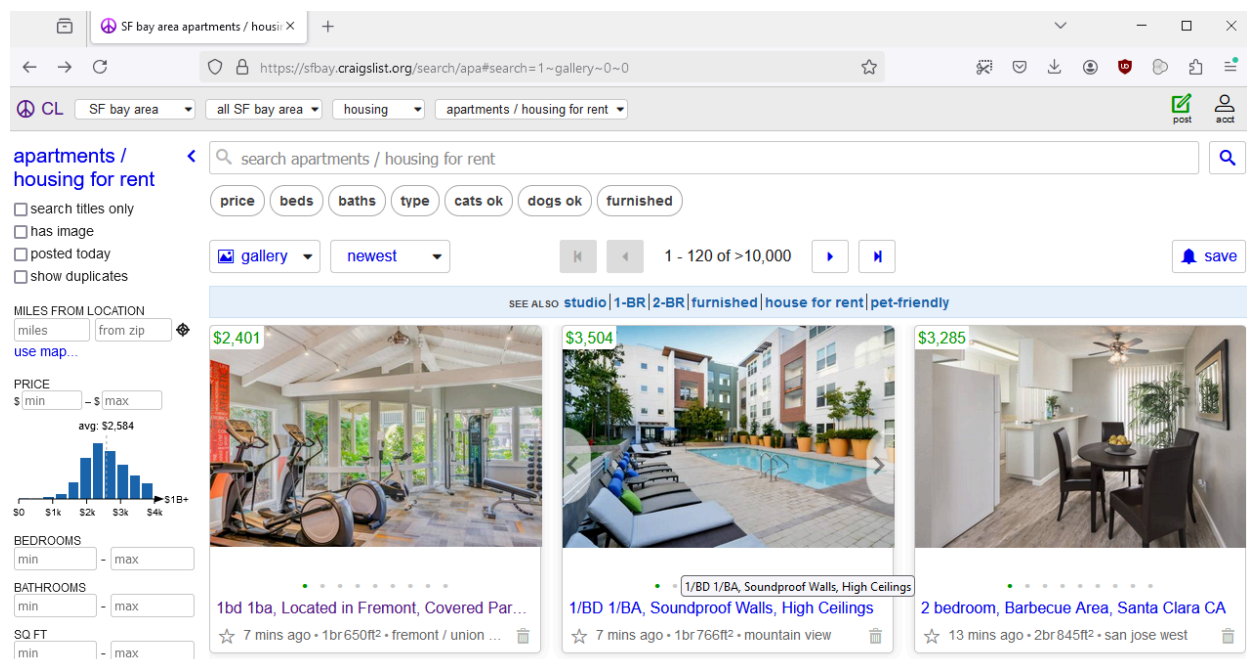
In the *Modeling* phase we trained 6 clustering models: KMeans, Hierarchical, Meanshift, Birch, DBSCAN, and HDBSCAN. This was followed by the *Evaluation* phase where we found that HDBSCAN had the highest silhouette score at 0.76, with the next best model (birch) at 0.57. It also had the lowest davies bouldin score at 0.32 with the next best model (birch) at 0.66. And lastly, although not directly comparable between models with different numbers of clusters it had the highest calinski harabasz score of ~65,000 with the next best model (birch) at ~11,000. HDBSCAN accomplished this by finding 206 clusters, and of our ~19,000 training data rows (after cleaning/pre-processing), only ~2000 were considered noise points.

Lastly, in the *Deployment* phase of CRISP-DM we train the model in Google Colab and deploy it to AWS S3. Our web application periodically checks every 5 minutes for model updates and automatically downloads/uses the latest one.

Therefore our end-to-end project is built on four key components: the frontend, backend, web scraper pipeline, and recommendation model training pipeline. The frontend, developed using React and Redux, ensures a responsive and modular user interface. The backend, powered by Python FastAPI, handles REST APIs, integrates with MongoDB for managing housing data, and utilizes Redis for caching frequently accessed information. The web scraper pipeline runs nightly on a cron and updates MongoDB with the latest rental listings, and the recommendation model training pipeline is manually kicked off in Google Colab.

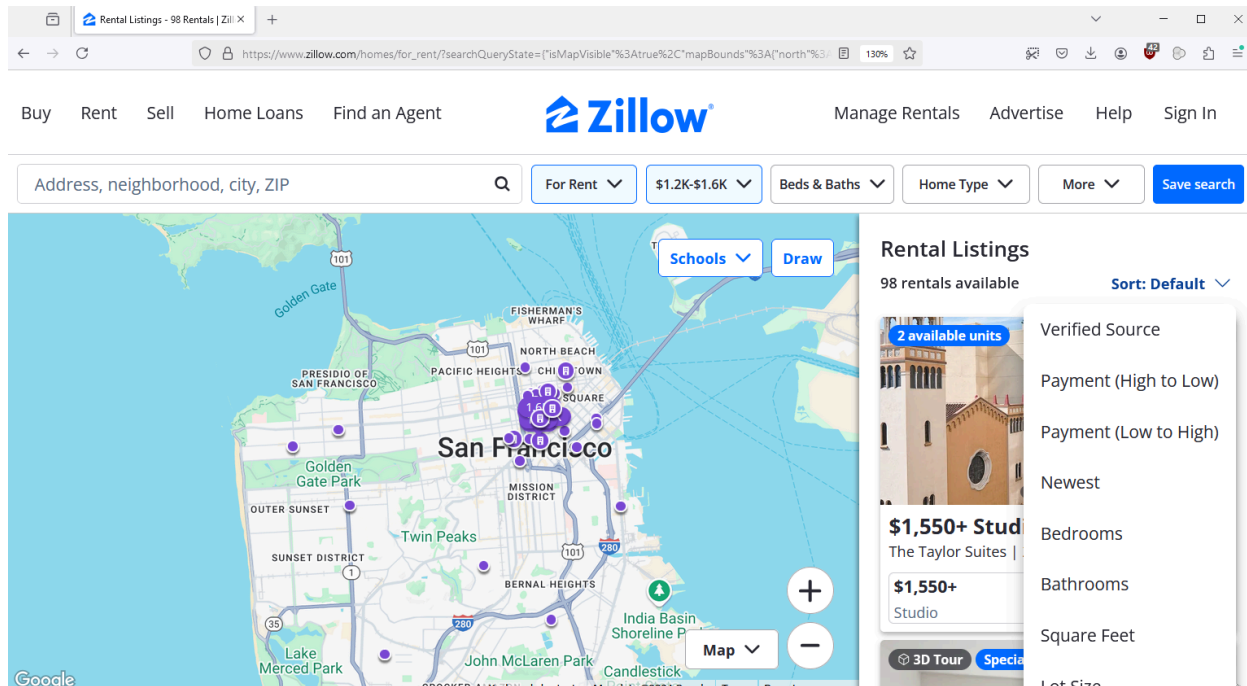
Related Work

Existing rental recommendation platforms, such as Zillow, realtor.com, and Craigslist, primarily rely on manual filters like dropdowns, checkboxes, and radio buttons to refine search results. Or alternatively they offer a map-based search where users can zoom in/out of the map and search for properties by location. While these methods are effective for basic filtering, scrolling through dozens of pages of property search results or clicking around on a map is less effective than being offered suggestions by the system based on similarity to current property would make finding a rental property easier.



Craigslist property search page

Some property search sites attempt to enhance the user experience with recommendation systems; however, these systems are typically implemented by the default sort pushing paid advertisements to the top.



Zillow property search page

Our system introduces a new approach by leveraging our clustering-based recommendation engine to dynamically refine property searches. Unlike existing platforms, our system is designed such that when a user explores a specific property, the system analyzes features like location, price, size, and amenities to recommend similar properties tailored to their preferences.

To build a rich and diverse rental listing database, we researched multiple data sources, including free APIs and web scraping from platforms like realtor.com, Zillow, Craigslist, and others. The scraped data is processed through our data cleaning and preprocessing pipeline to ensure that each data source maps to the same data schema in the database.

Data

- **Source:** Data has been scraped from Realtor.com using a python script.
- **Type and Structure:** Raw data that is scraped is structured as follows:

Category	Field	Data Type	Description
Basic Information	property_url	string	URL link to the property's listing page
	property_id	string	Unique identifier for the property
	listing_id	string	Identifier for the property listing

	mls	string	Multiple Listing Service name
	mls_id	string	Unique ID in the MLS system
	status	string	Current listing status (e.g., active, pending, sold)
Address Details	street	string	Street address of the property
	unit	string	Unit number or apartment number
	city	string	City where the property is located
	state	string	State abbreviation
	zip_code	string	ZIP or postal code
Property Description	style	string	Architectural style of the property
	beds	integer	Number of bedrooms
	full_baths	integer	Number of full bathrooms
	half_baths	integer	Number of half bathrooms
	sqft	integer	Total square footage of the property
	year_built	integer	Year the property was built
	stories	integer	Number of stories (floors) in the property
	garage	integer	Number of garage spaces
	lot_sqft	integer	Square footage of the lot
Property Listing Details	days_on_mls	integer	Number of days the property has been on MLS
	list_price	number	Current listing price
	list_price_min	number	Minimum listing price range
	list_price_max	number	Maximum listing price range
	list_date	date	Date the property was listed
	pending_date	date	Date the property status changed to pending
	sold_price	number	Final sale price of the property
	last_sold_date	date	Date of the last sale transaction
	price_per_sqft	number	Listing price per square foot
	new_construction	boolean	Indicates if the property is newly constructed
	hoa_fee	number	Homeowners Association fee
Location Details	latitude	number	Latitude coordinate of the property

	longitude	number	Longitude coordinate of the property
	nearby_schools	array(string)	List of nearby schools
Agent Info	agent_id	string	Unique identifier for the agent
	agent_name	string	Name of the agent
	agent_email	string	Email address of the agent
	agent_phone	string	Contact phone number of the agent
Broker Info	broker_id	string	Unique identifier for the broker
	broker_name	string	Name of the brokerage
Builder Info	builder_id	string	Unique identifier for the builder
	builder_name	string	Name of the builder
Office Info	office_id	string	Unique identifier for the office
	office_name	string	Name of the real estate office
	office_phones	array(string)	List of phone numbers for the office
	office_email	string	Email address for the office

- **Cleaning and Preprocessing:** Steps taken to clean and prepare the data, such as:

Step	Action Taken	Explanation
Removing Duplicates	Dropped duplicate rows based on property_id.	Some rows were duplicated due to multiple scrapes of the same properties. And since the property Id is the primary key hence these duplicates were removed based on property Id
Handling Missing Data	Imputed missing values for certain columns (e.g., beds, full_baths, half_baths, stories, parking_garage, hoa_fee) like setting the value of missing half-baths to 0 indicating that half-baths do not exist in that apartment.	Columns with excessive missing data like half-baths, stories were either imputed or dropped.
Dropping Irrelevant Columns	1) Some columns like IDs, contact details, agent names, seller contacts and details do not contribute much to the price of the rental property and were	We used the correlation_matrix and other EDA tools to understand which columns were most correlated to each other and which columns did not have

	hence dropped. 2) On the other hand, highly correlated columns like neighbourhoods, nearby-schools were feature engineered to make one column	any impact on the prices
Text Field Normalization	Cleaned and standardized text fields. For example status was filtered to remove irrelevant values	The status column had values like "FOR_RENT", "CONTINGENT" and "PENDING" which added to the noise but did not contribute to price prediction. Such noise was processed to focus on relevant categories
Numerical Field Transformation	Transformed the categorical data to numerical values using one-hot encoding in order to be able to feed into the model	

- **Data Volume:** We began with data that had 22418 rows and 57 columns.
- A total of 19114 rows and 28 columns remained after most of the pre-processing was done.. We then used PCA, UMAP as well as autoencoder to further reduce the dimensionality as well as compare the results from all 3 techniques. PCA reduced the number of columns down to 25 and Autoencoder/UMAP both reduced them down to 10.

Methods

Frontend

The frontend is written in React, and each section of the page has React components. For example, RentalListings, IndividualRentalListing, and RentalRecommendations. These components share state through Redux, which propagates state changes via events. This keeps each component isolated and encapsulated.

Backend

The backend web services are written in Python using the FastAPI framework for REST APIs and Websockets. MongoDB is used for the No-SQL housing data, and Postgres is used for user data. Redis caching is used to keep the rental property details context in the chat service to avoid re-querying listing-service to fetch housing data from MongoDB.

Web Scraper Pipeline

The web scraper runs on a cron nightly and scrapes data from sites such as realtor.com and zillow.com. The scraped pages/data is sent to a Kafka queue which is consumed by the data-sink service. The data-sink uses Apache Flink to stream these scraped items and performs some data cleaning/preprocessing before saving the cleaned housing data in MongoDB.

Recommendation Model and Training Pipeline

The recommendation model is used to suggest similar listings to the one(s) the user is looking at. This model is trained offline in a Google Colab. The model training data is read from MongoDB and the trained model is uploaded to AWS s3.

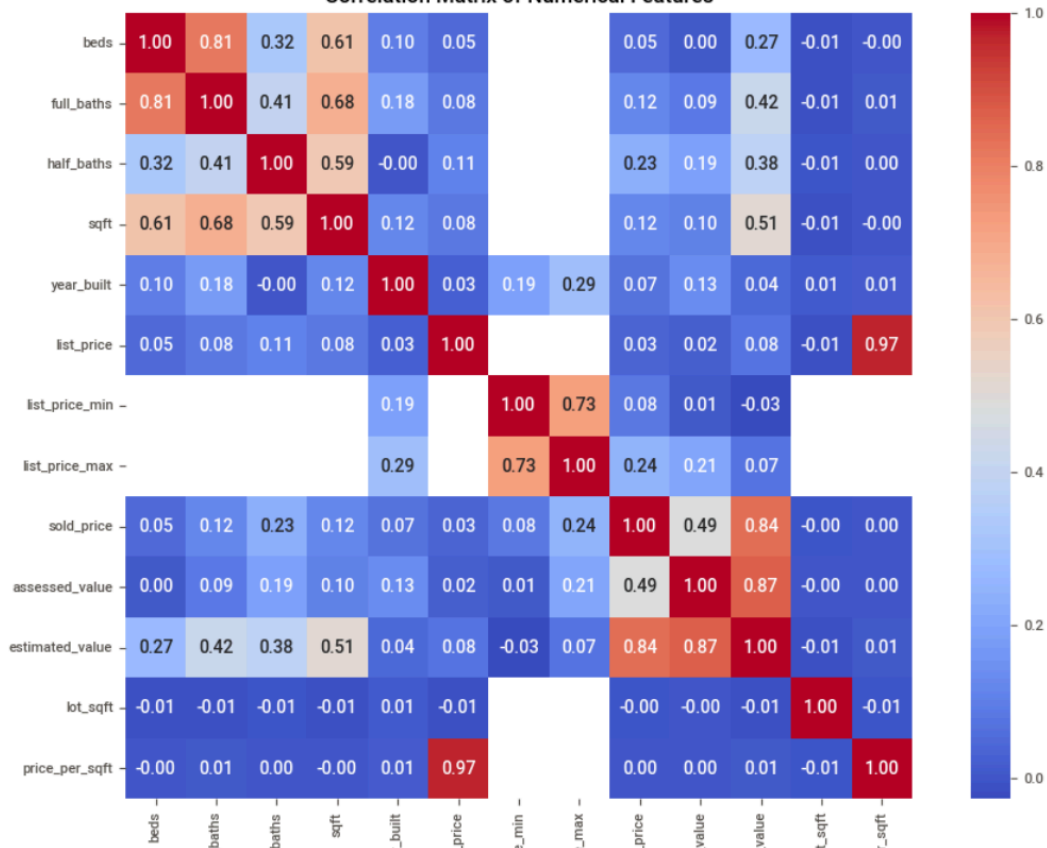
To develop our recommendation model, we followed a comprehensive data preparation and analysis process to ensure the model's accuracy and relevance. The training pipeline involved the following steps:

1. Data Loading and Exploration:

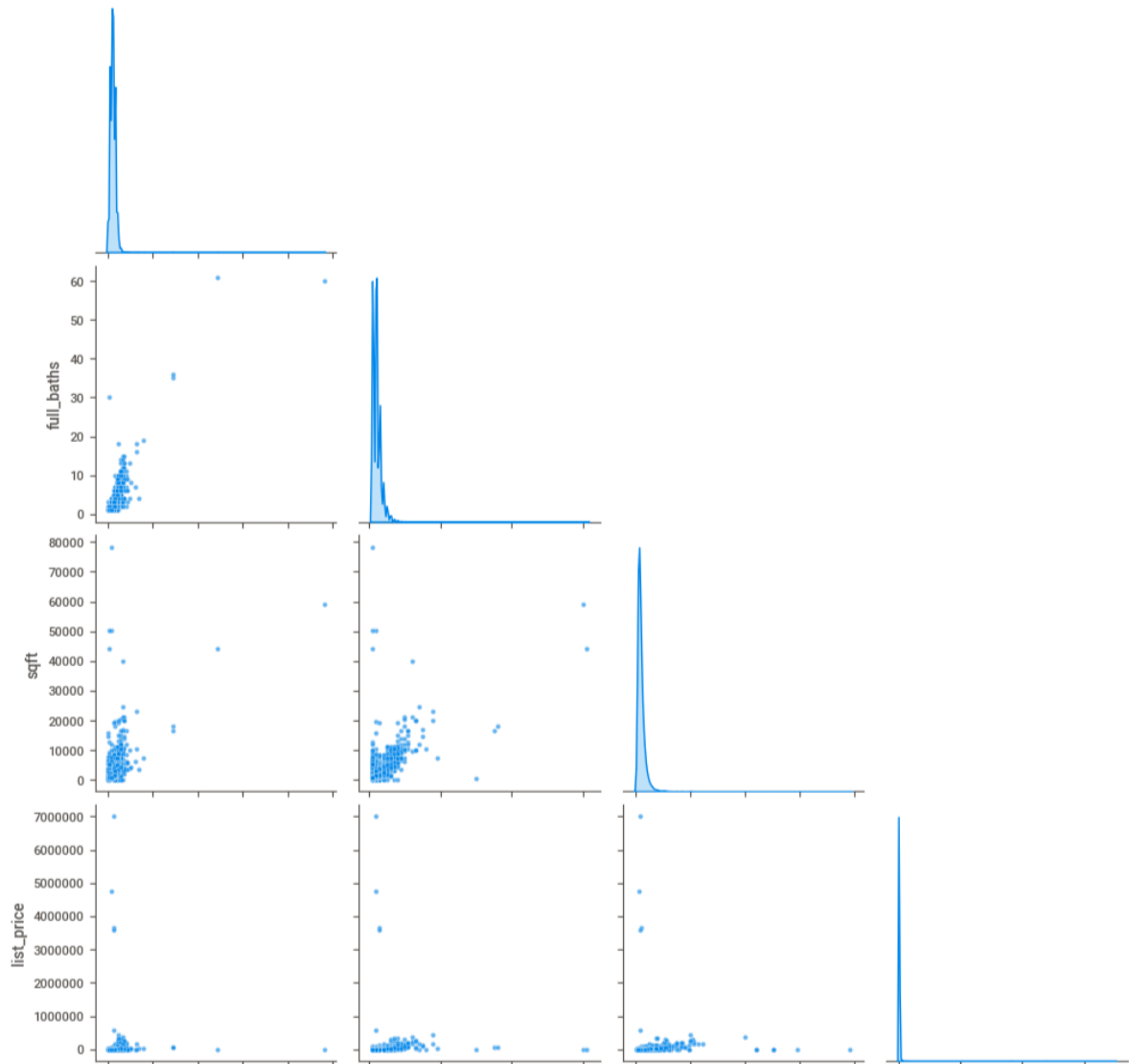
The process began by loading a housing dataset containing key features such as location, size, price, and amenities. Exploratory Data Analysis (EDA) was conducted to understand the dataset's structure, identify missing values, and detect outliers. Visualizations such as heatmaps, scatterplots, geographical distributions, time series analyses, and box plots provided valuable insights into data patterns. A sample of our EDA plots:

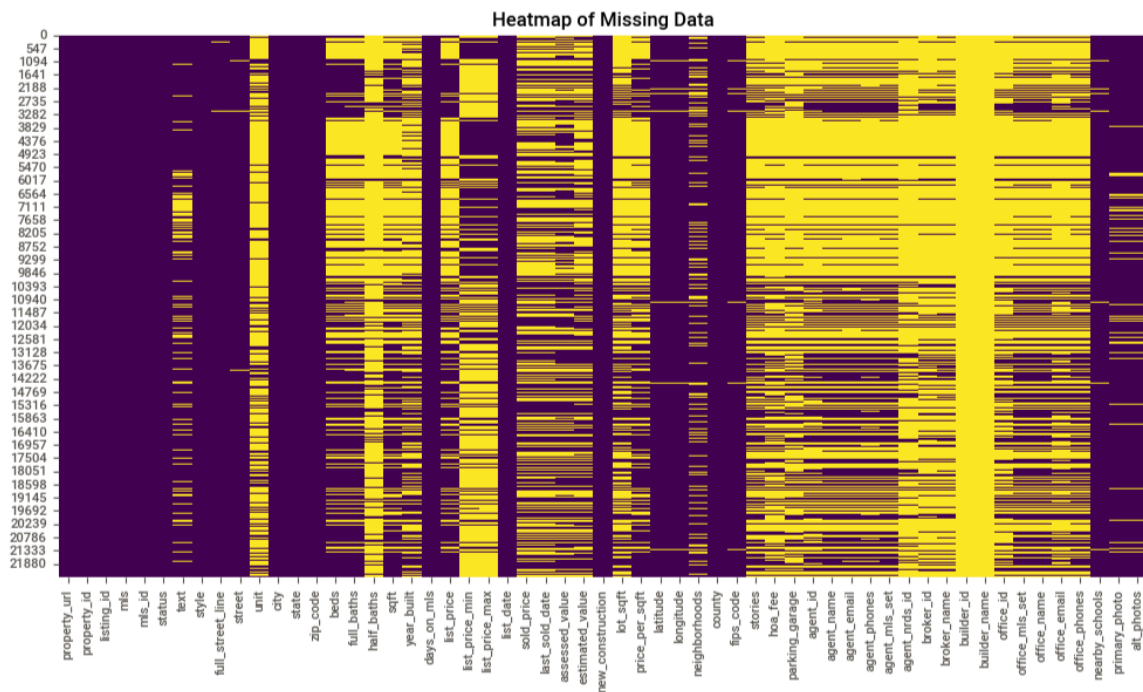


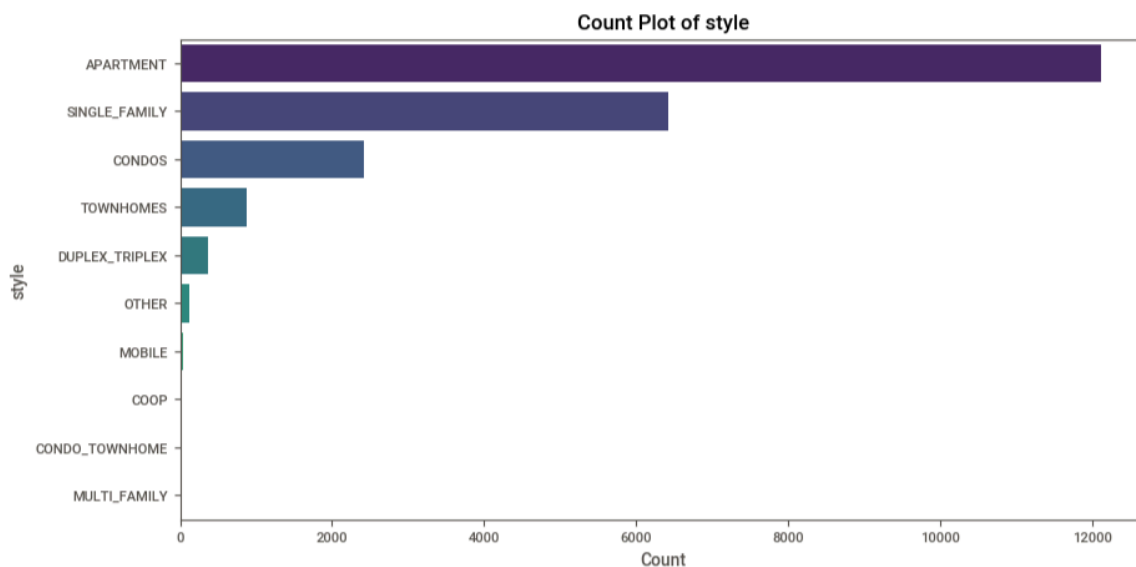
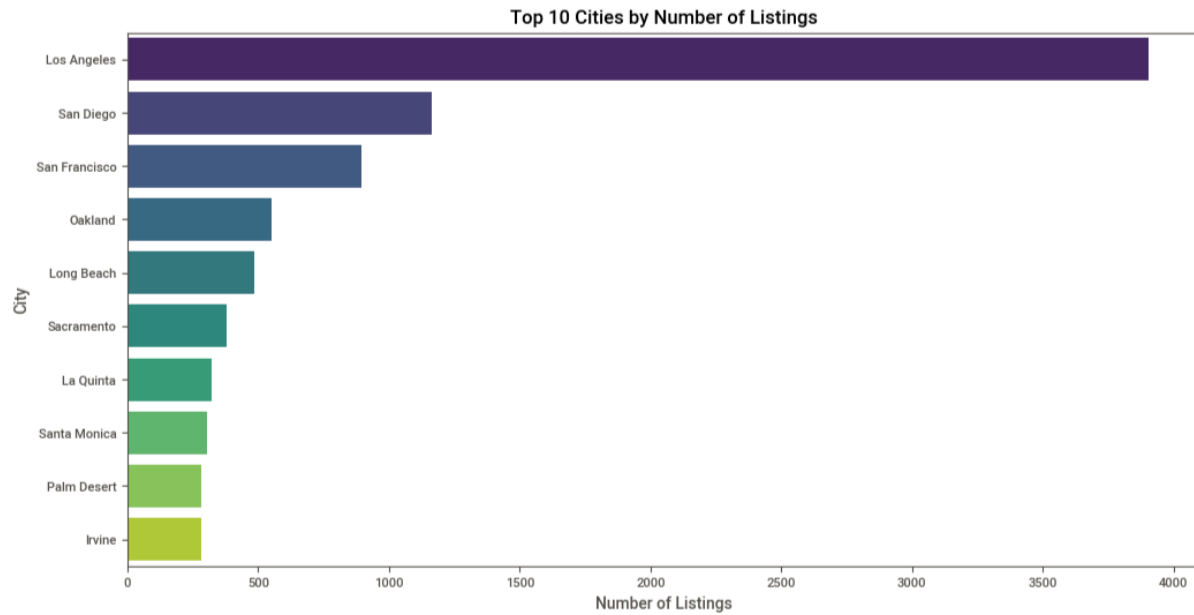
Correlation Matrix of Numerical Features



Pairwise Relationships Between Numerical Features







2. Data Cleaning and Preprocessing:

We addressed data quality issues by handling duplicates, standardizing text fields, converting data types, and imputing missing values using the Iterative Imputer (MICE). Outliers were detected and removed to improve the data's consistency and reliability. For example, here's our custom imputation and outlier removal code:

▼ Imputation

Imputation to handle missing or irregular data.

```
▶ ### Imputation for Missing Values

# Impute `beds` with 1 for missing and zeroed values.
df['beds'] = df['beds'].fillna(1)
df.loc[df['beds'] == 0.0, 'beds'] = 1.0

# Impute `full_baths` with 1 for missing values.
df['full_baths'] = df['full_baths'].fillna(1)

# Impute `half_baths` with 0 for missing values.
df['half_baths'] = df['half_baths'].fillna(0)

# Impute `full_baths` with 1 if both `full_baths` and `half_baths` are 0
df.loc[(df['full_baths'] == 0.0) & (df['half_baths'] == 0.0), 'full_baths'] = 1.0

# Impute `hoa_fee` with 0 for missing values.
df['hoa_fee'] = df['hoa_fee'].fillna(0)

# Impute `stories` with 1 for missing values.
df['stories'] = df['stories'].fillna(1)

# Impute `parking_garage` with 0 for missing values.
df['parking_garage'] = df['parking_garage'].fillna(0)

# Impute `year_built` using last_sold_date
df['year_built'] = df['year_built'].fillna(df['last_sold_date'].dt.year)
```

▼ Outliers Removal

Filtering data to eliminate outliers by flagging data that needs to be removed.

This needs to occur before MICE Imputation below otherwise imputed values will be wildly wrong.

```
invalid_flags = [
    'is_invalid_beds',
    'is_invalid_full_baths',
    'is_invalid_half_baths',
    'is_invalid_year_built',
    'is_invalid_sqft',
    'is_invalid_lot_sqft',
    'is_invalid_list_date',
]

def detect_outliers(df):
    # Flagging invalid Bedrooms and Bathrooms
    df['is_invalid_beds'] = ~df['beds'].between(0, 10, inclusive="both")
    df['is_invalid_full_baths'] = ~df['full_baths'].between(0, 10, inclusive="both")
    df['is_invalid_half_baths'] = ~df['half_baths'].between(0, 10, inclusive="both")

    # Flagging invalid Year Built
    current_year = datetime.now().year
    df['is_invalid_year_built'] = ~df['year_built'].between(1800, current_year, inclusive="both")

    # Flagging invalid Lot Size
    df['is_invalid_sqft'] = ~(df['sqft'].between(0, 20_000, inclusive="both") | df['lot_sqft'].between(0, 20_000, inclusive="both"))
    df['is_invalid_lot_sqft'] = ~df['lot_sqft'].between(0, 20_000, inclusive="both")

    # Flagging invalid listing dates
    current_date = datetime.now()
    df['is_invalid_list_date'] = df['list_date'] > current_date

    df['is_invalid_row'] = df[invalid_flags].any(axis=1)

    print(f"Invalid beds: {df['is_invalid_beds'].sum()}")
    print(f"Invalid full_baths: {df['is_invalid_full_baths'].sum()}")
    print(f"Invalid half_baths: {df['is_invalid_half_baths'].sum()}")
    print(f"Invalid year_built: {df['is_invalid_year_built'].sum()}")
    print(f"Invalid sqft: {df['is_invalid_sqft'].sum()}")
    print(f"Invalid lot_sqft: {df['is_invalid_lot_sqft'].sum()}")
    print(f"Invalid list_date: {df['is_invalid_list_date'].sum()}")

    return df
```

3. Feature Engineering:

To enrich the dataset, we created new features, including calculated attributes like total bathrooms and categorical attributes like year-built categories. Additional features like has_pool and has_air_conditioning were derived using natural language processing (NLP) on property descriptions. Features were one-hot encoded into numerical representations, and numerical values were scaled using MinMaxScaler for uniformity. For example, here we create new features and perform one-hot encoding:

✓ Feature Engineering

(Feature creation) Creating new feature: **total_bath**

```
print(df.columns) # Print all column names

Index(['text', 'style', 'city', 'zip_code', 'beds', 'full_baths', 'half_baths',
       'sqft', 'year_built', 'list_price', 'list_date', 'sold_price',
       'last_sold_date', 'assessed_value', 'estimated_value',
       'new_construction', 'lot_sqft', 'price_per_sqft', 'county', 'stories',
       'hoa_fee', 'parking_garage'],
      dtype='object')

[ ] df['total_baths'] = df['full_baths'] + (df['half_baths'] * 0.5) # half_baths contribute less
df = df.drop(columns=['full_baths', 'half_baths'], errors='ignore')

df.columns

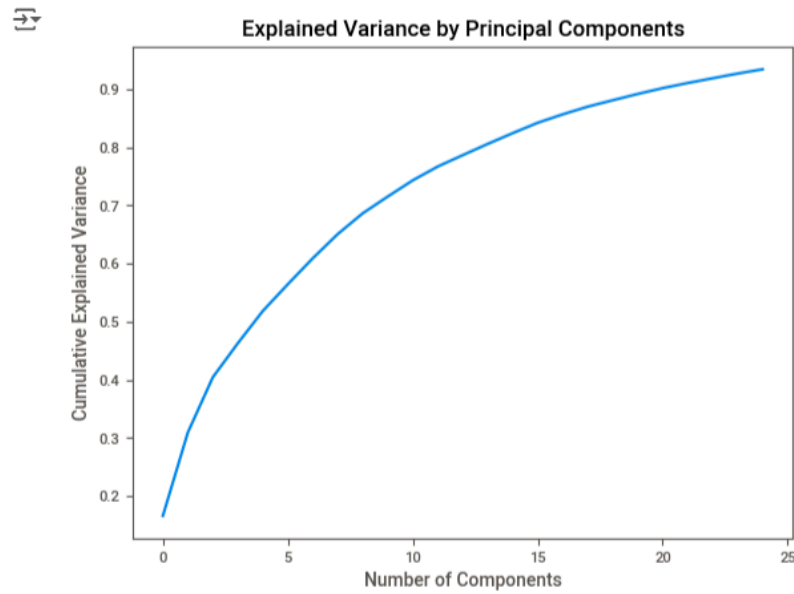
Index(['text', 'style', 'city', 'zip_code', 'beds', 'sqft', 'year_built',
       'list_price', 'list_date', 'sold_price', 'last_sold_date',
       'assessed_value', 'estimated_value', 'new_construction', 'lot_sqft',
       'price_per_sqft', 'county', 'stories', 'hoa_fee', 'parking_garage',
       'total_baths'],
      dtype='object')

# One-hot encoding:
categorical_columns = ['style', 'county']
encoded_df = pd.get_dummies(df, columns=categorical_columns, prefix=categorical_columns, drop_first=True)
encoded_df.info()
```

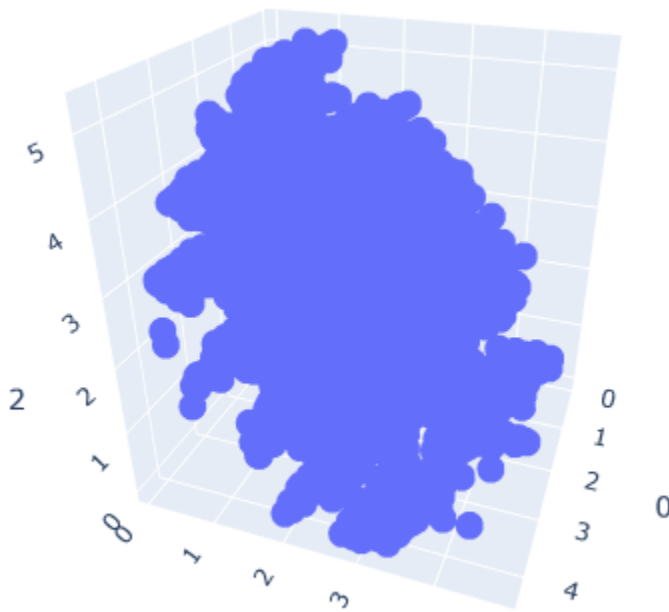
4. Dimensionality Reduction:

Dimensionality reduction techniques such as PCA were initially explored, but due to non-linear patterns in the data we ultimately ended up 1) training our own Autoencoder using tensorflow and 2) using UMAP. UMAP demonstrated better performance in reducing dimensions while preserving data patterns over the Autoencoder. Unfortunately while the Autoencoder had very low error in performing encoding, it didn't keep the data points separated in a lower dimension and all the clustering algorithms struggled to find well-separated clusters. Dimensionality reduction also enabled visualizations of clusters of 3D, helping to uncover distinct property groups and insights about shared characteristics within clusters.

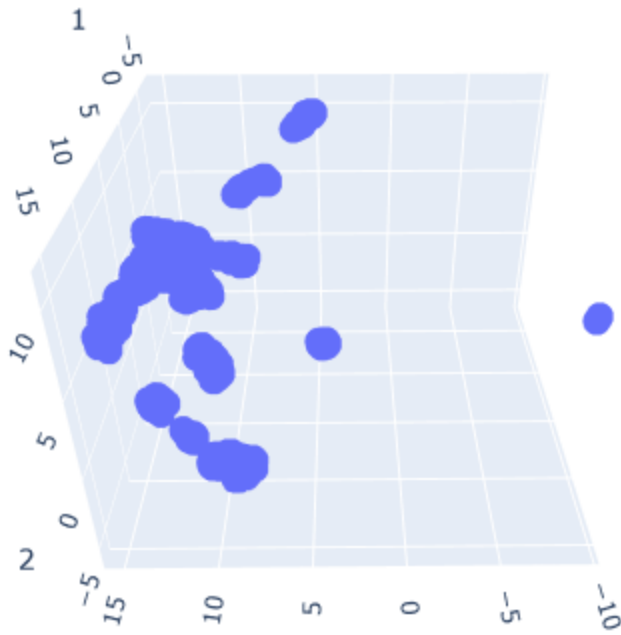
First we tried PCA, but the elbow plot doesn't make it to 0.9 until the number of components is between 15-20.



Then we tried an Autoencoder trained using tensorflow. But when visualizing it (in 3 dimensions) it doesn't separate the data into clusters well:



Lastly we tried UMAP which separated the data excellently, there are many distinct clusters:



Therefore, we used the UMAP dimensionality reduced data to perform clustering.

5. Clustering Algorithms:

We applied clustering techniques such as K-Means, Hierarchical, Meanshift, Birch, DBSCAN, and HDBSCAN clustering to group similar properties based on their features. These algorithms were selected as scikit-learn has built-in support for most of these. We also attempted to use Gaussian Mixture and Spectral clustering, however, during training these algorithms had floating point underflow errors on our dataset.

```

import hdbscan
from scipy.cluster.hierarchy import dendrogram, linkage, fcluster
from sklearn.cluster import Birch, DBSCAN, KMeans, MeanShift, MiniBatchKMeans, SpectralClustering
from sklearn.mixture import GaussianMixture

def birch_clustering(data, n_clusters=5):
    birch_model = Birch(n_clusters=n_clusters)
    labels = birch_model.fit_predict(data)
    return birch_model, labels

def kmeans_clustering(data, n_clusters=5):
    kmeans_model = KMeans(n_clusters=n_clusters, random_state=42)
    labels = kmeans_model.fit_predict(data)
    return kmeans_model, labels

def meanshift_clustering(data, n_clusters=5):
    meanshift_model = MiniBatchKMeans(n_clusters=n_clusters, random_state=42)
    labels = meanshift_model.fit_predict(data)
    return meanshift_model, labels

def minibatch_kmeans_clustering(data, n_clusters=5):
    minibatch_kmeans_model = MiniBatchKMeans(n_clusters=n_clusters, random_state=42)
    labels = minibatch_kmeans_model.fit_predict(data)
    return minibatch_kmeans_model, labels

def dbscan_clustering(data, min_samples=15, eps=0.1):
    dbscan_model = DBSCAN(min_samples=min_samples, eps=eps)
    labels = dbscan_model.fit_predict(data)
    return dbscan_model, labels

def hdbscan_clustering(data, min_samples=15, min_cluster_size=15, prediction_data=False):
    hdbscan_model = hdbscan.HDBSCAN(min_samples=min_samples, min_cluster_size=min_cluster_size, prediction_data=prediction_data)
    labels = hdbscan_model.fit_predict(data)
    return hdbscan_model, labels

def hierarchical_clustering(data, method='ward', n_clusters=5):
    Z = linkage(data, method=method)
    labels = fcluster(Z, t=n_clusters, criterion='maxclust')
    return Z, labels

```

6. Hyperparameter Optimization:

We computed inertia, silhouette, davies bouldin, and calinski harabasz scores. Since we do not have labeled data we do not have scores such as accuracy, precision, recall, etc. For every clustering algorithm that required providing the number of clusters upfront we trained 19 models (num_clusters=2 through 20) and plotted charts measuring these scores. For DBSCAN we tried all combinations of eps=0.1 through 0.9 and min_samples=10 through 50. For HDBSCAN we tried all combinations of min_cluser_size=10 through 50, and min_samples=10 through 50. The final hyperparameters were chosen based on which parameters performed best across all scores.

```

from sklearn.metrics import silhouette_samples, silhouette_score, davies_bouldin_score, calinski_harabasz_score

def compute_clustering_metrics(df,
                               cluster_function,
                               compute_inertia_scores=False,
                               compute_silhouette_scores=True,
                               compute_davies_bouldin_scores=True,
                               compute_calinski_harabasz_scores=True):

    inertia_scores = []
    silhouette_scores = []
    davies_bouldin_scores = []
    calinski_harabasz_scores = []

    for num_clusters in range(2, 20):
        print(f"Computing Metrics for num_clusters={num_clusters}")

        model, labels = cluster_function(df, n_clusters=num_clusters)

        if compute_inertia_scores:
            inertia_scores.append(model.inertia_)
        if compute_silhouette_scores:
            silhouette_scores.append(silhouette_score(df, labels))
        if compute_davies_bouldin_scores:
            davies_bouldin_scores.append(davies_bouldin_score(df, labels))
        if compute_calinski_harabasz_scores:
            calinski_harabasz_scores.append(calinski_harabasz_score(df, labels))

    return inertia_scores, silhouette_scores, davies_bouldin_scores, calinski_harabasz_scores

```

7. Deployment

We created a model training pipeline using a Google colab notebook which uploads our trained recommendations model to AWS s3 storage. We use the boto3 Python library to download/upload objects to s3, here's how we download the latest model in the s3 bucket:

```

import boto3
s3_client = boto3.client('s3')

def get_latest_file():
    response = s3_client.list_objects_v2(Bucket='rentalrecommender')
    contents = response["Contents"]
    latest_file = max(contents, key=lambda x: x['LastModified'])
    return latest_file

```

The rental-recommendation service periodically polls s3 to see if a new model has been uploaded, and if so automatically downloads and starts using it.

```

CHECK_INTERVAL_SECONDS = 300 # 5 mins

async def download_latest_model():
    while True:
        # Get the latest file in S3 bucket

```

```

latest_file = get_latest_file()
if latest_file:
    latest_file_version = latest_file["ETag"]
    if latest_file_version != current_model_version:
        s3_client.download_file(
            BUCKET_NAME,
            latest_file["Key"],
            str(TEMP_FILE_PATH),
        )

        os.replace(TEMP_FILE_PATH, CURRENT_FILE_PATH)
        current_model_version = latest_file_version
        print("New model downloaded.")
    else:
        print("Current model is up-to-date.")
await asyncio.sleep(CHECK_INTERVAL_SECONDS)

```

Here's the Google colab code to upload the trained model to s3 storage:



+ Code + Text



Save Model to Disk



```
[ ] import pickle

MODEL_FILENAME = "clustering_model.pkl"

# Save model to disk in pickled format
with open(MODEL_FILENAME, "wb") as f:
    pickle.dump(kmeans_model, f)
```

Upload Model to AWS s3 Storage

```
[ ] from getpass import getpass

AWS_ACCESS_KEY_ID = getpass()
AWS_SECRET_ACCESS_KEY = getpass()
AWS_DEFAULT_REGION = "us-east-1"
```



```
.....
.....
```



```
import boto3

BUCKET_NAME = "rentalrecommender"

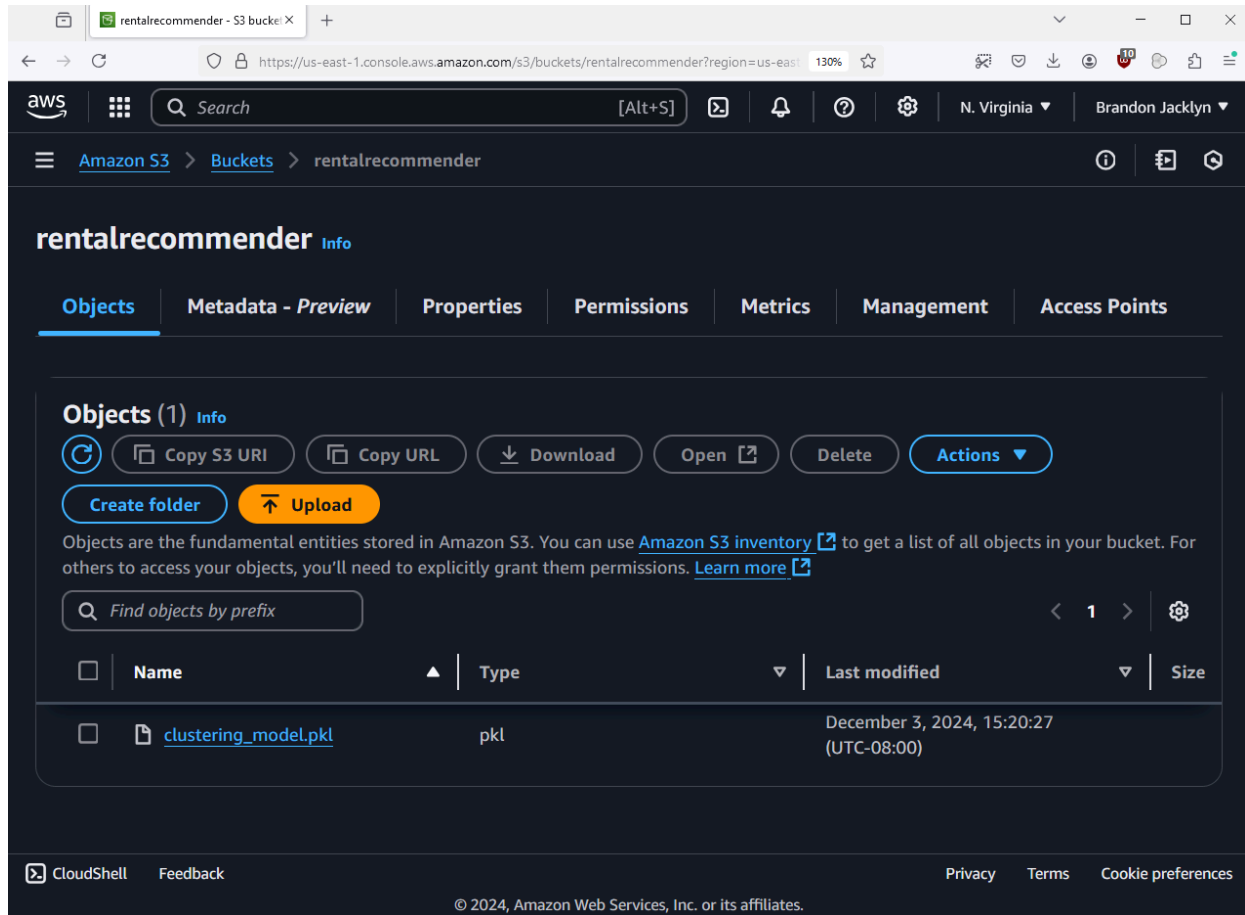
s3 = boto3.client(
    's3',
    aws_access_key_id=AWS_ACCESS_KEY_ID,
    aws_secret_access_key=AWS_SECRET_ACCESS_KEY,
    region_name=AWS_DEFAULT_REGION,
)

try:
    s3.upload_file(MODEL_FILENAME, BUCKET_NAME, MODEL_FILENAME)
    print(f"File {MODEL_FILENAME} uploaded to {BUCKET_NAME}/{MODEL_FILENAME}")
except Exception as e:
    print(f"Error uploading file: {e}")
```



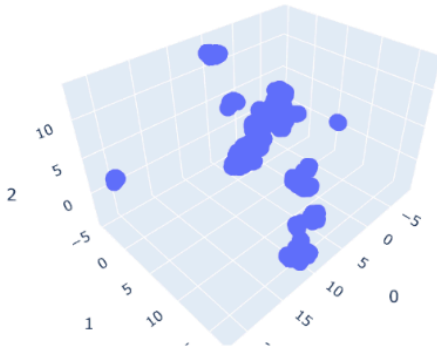
```
File clustering_model.pkl uploaded to rentalrecommender/clustering_model.pkl
```

And here's the s3 storage for our clustering model:

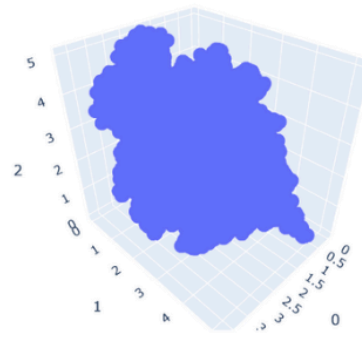


Experiments and Results

To achieve the best results possible, we compared the performance of our data preprocessing method using UMAP, PCA, and an Autoencoder for dimensionality reduction. We found that UMAP outperformed both PCA and the autoencoder providing a clearer separation of clusters and a more meaningful representation of the data. The results indicate that UMAP preserves both local and global structures, leading to better clustering outcomes compared to PCA, which failed to capture complex relationships in the data, and the autoencoder, which showed inconsistent results due to its overfitting tendency.

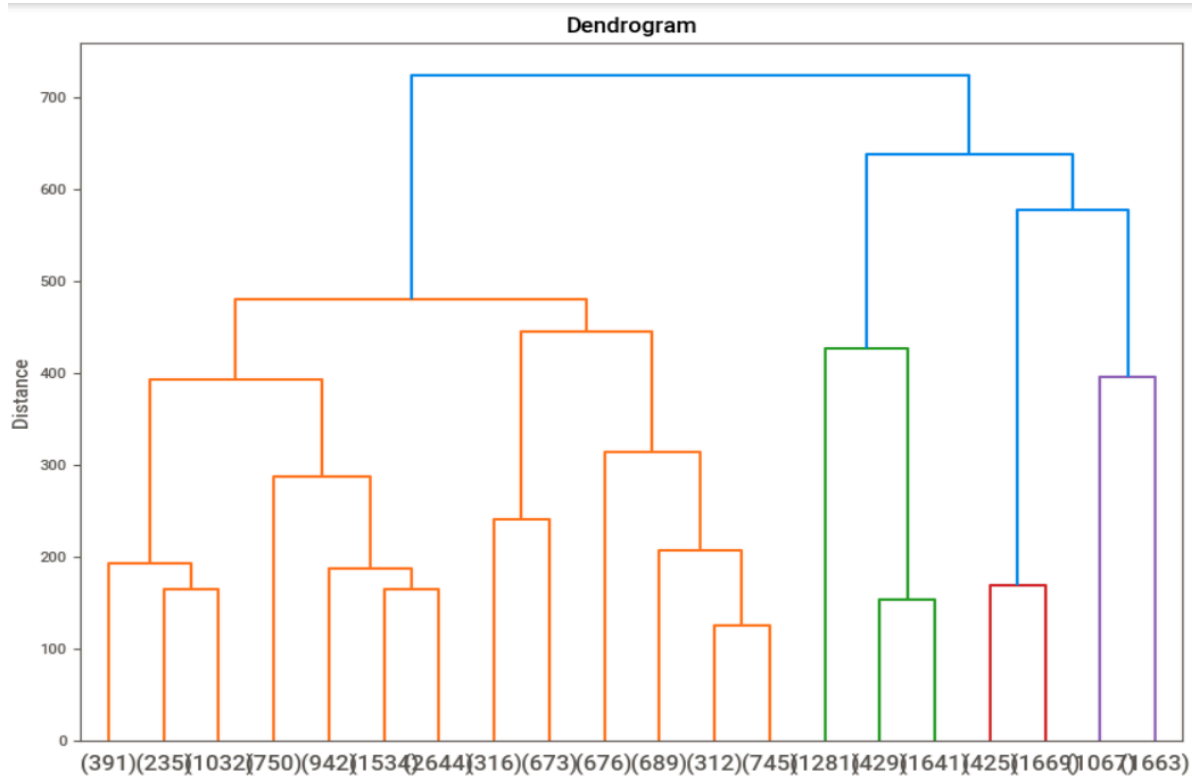


Results from UMap

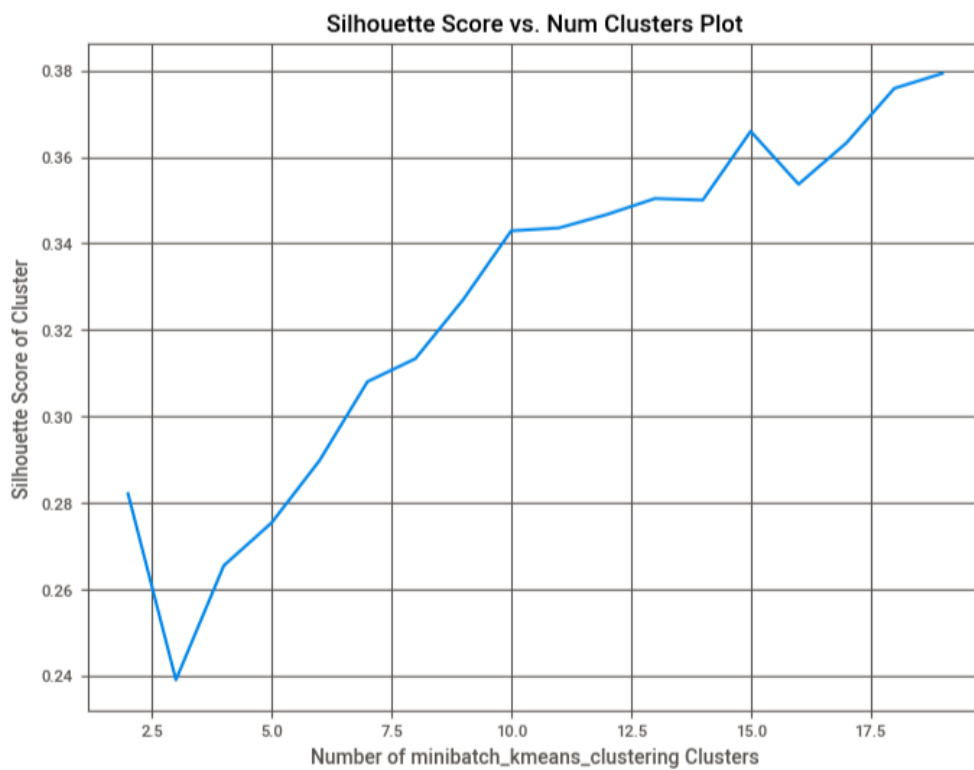
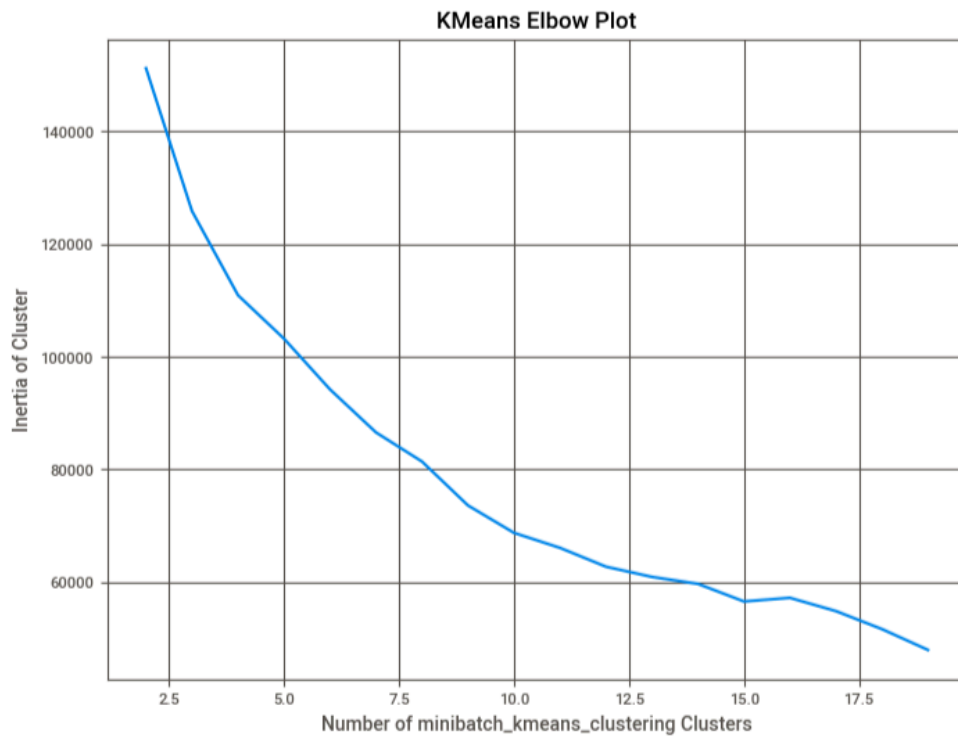


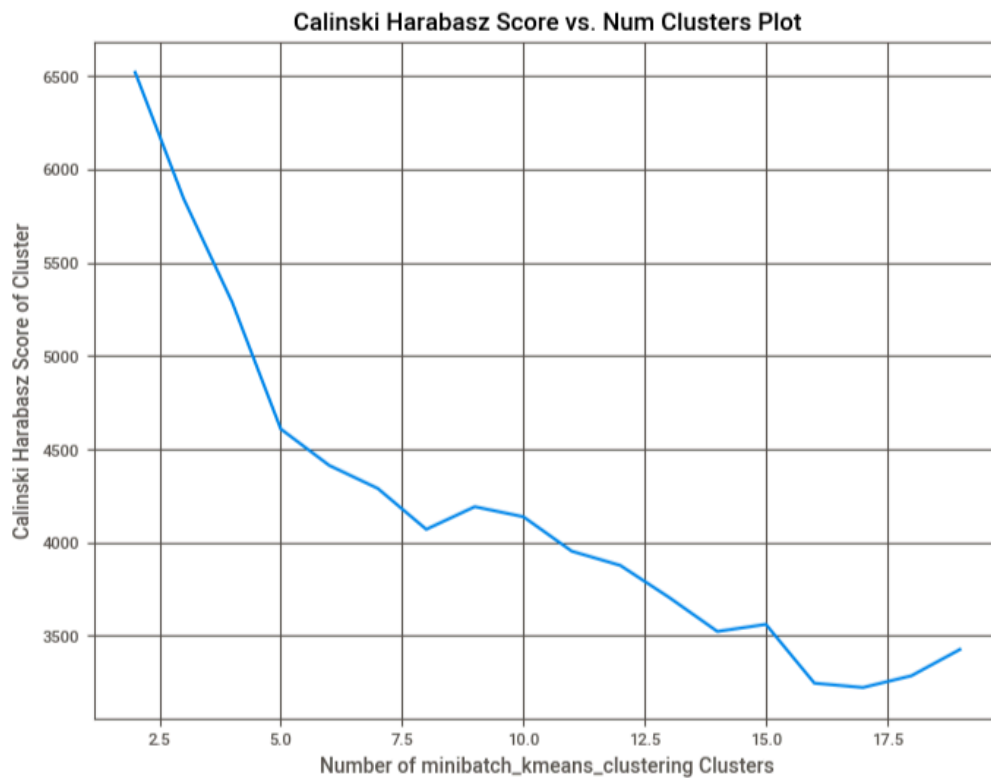
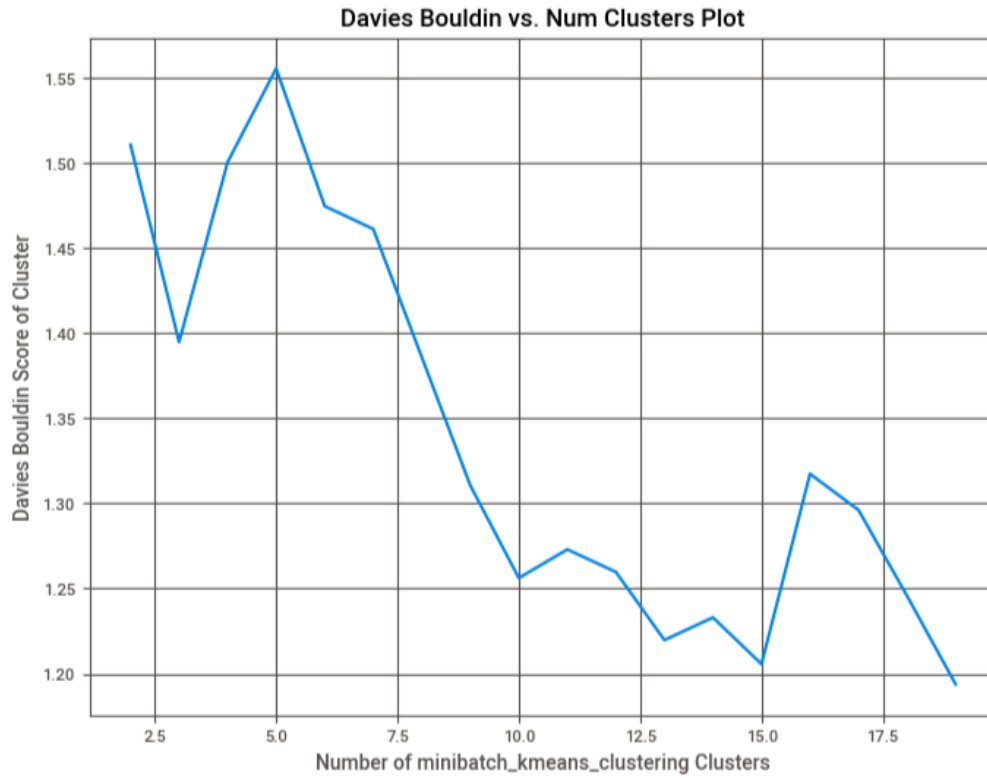
Results for Autoencoder

For the hierarchical clustering we were able to plot the dendrogram:

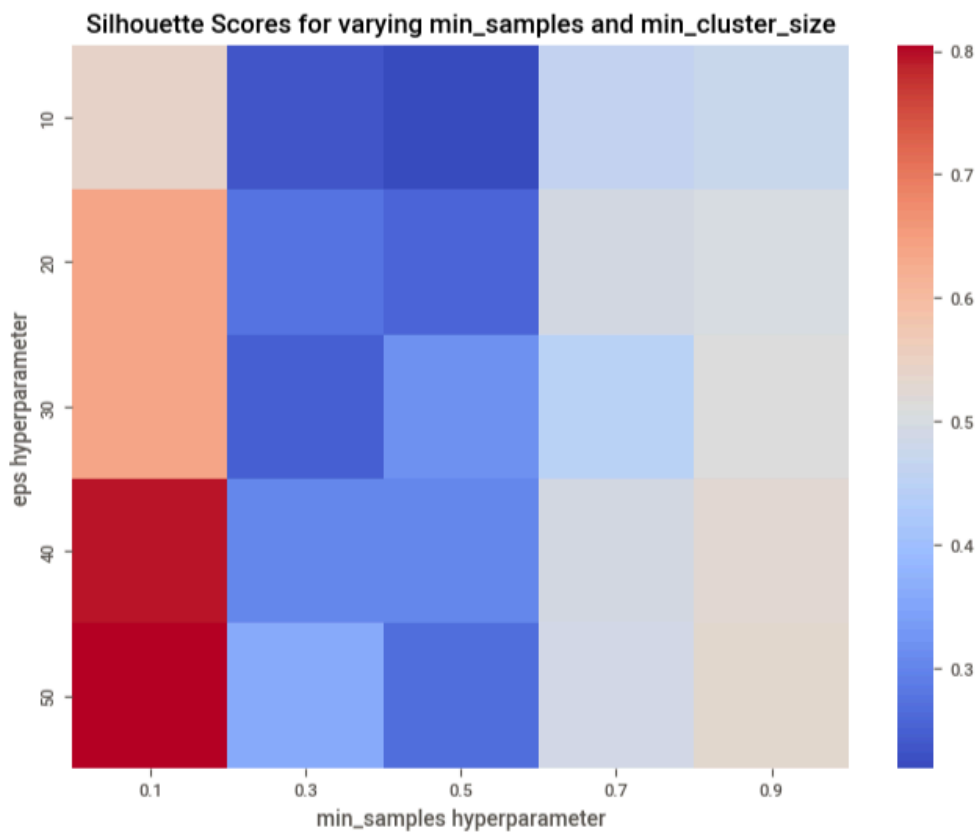


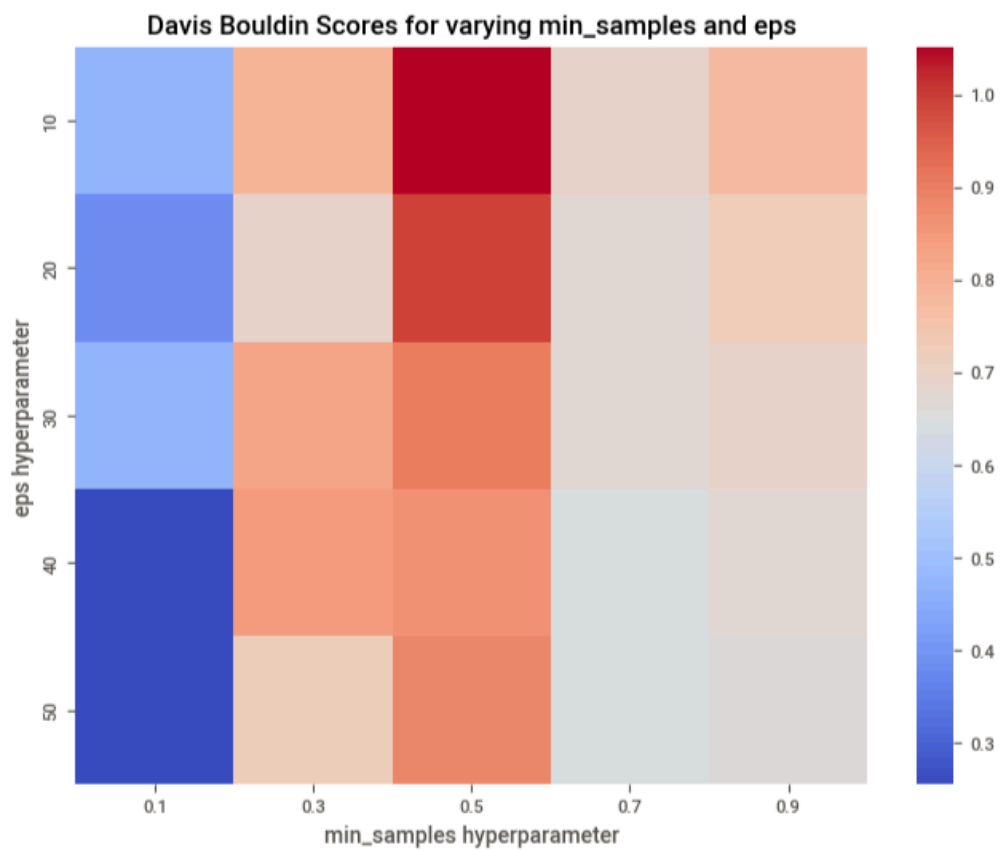
For each of the 6 clustering algorithms we performed hyperparameter optimization to plot out these 3 scores: Silhouette score, Davies Bouldin score and Calinski Harabasz score. For example, here are the score plots for K-Means on the UMAP reduced dataset where the scores are plotted on the y-axis for each given num_clusters=N on the x-axis. We have these same plots for hierarchical, birch, and meanshift clustering as well.

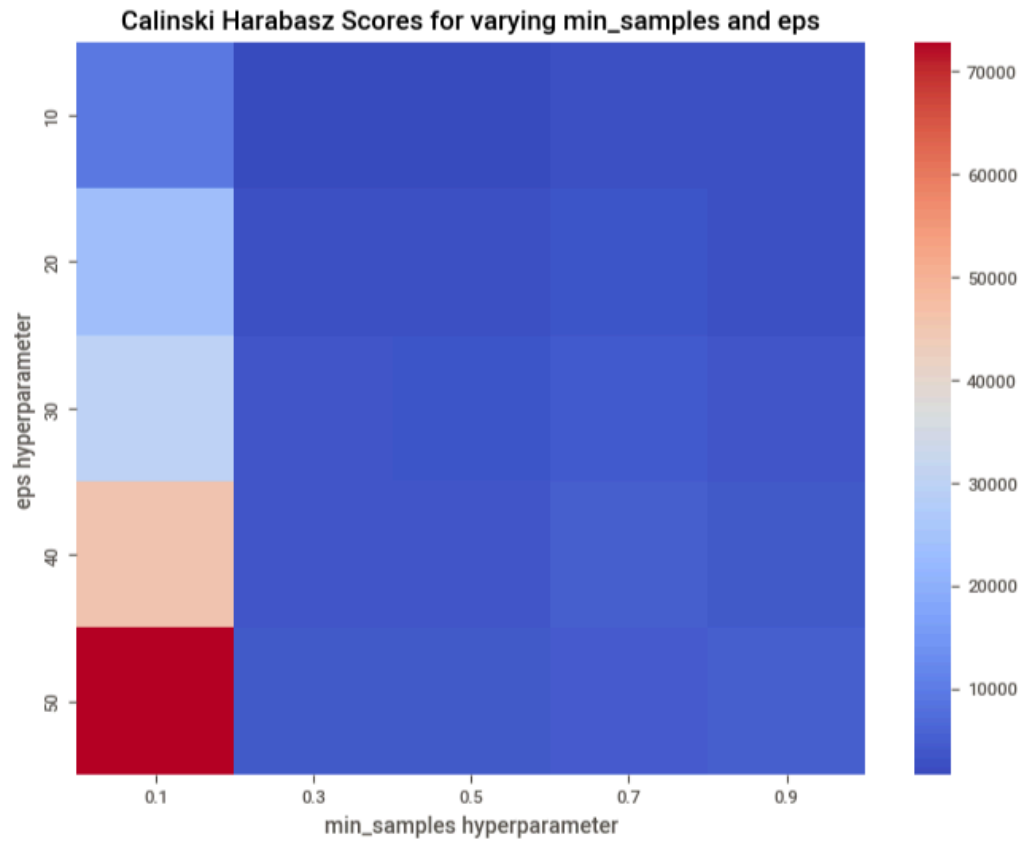




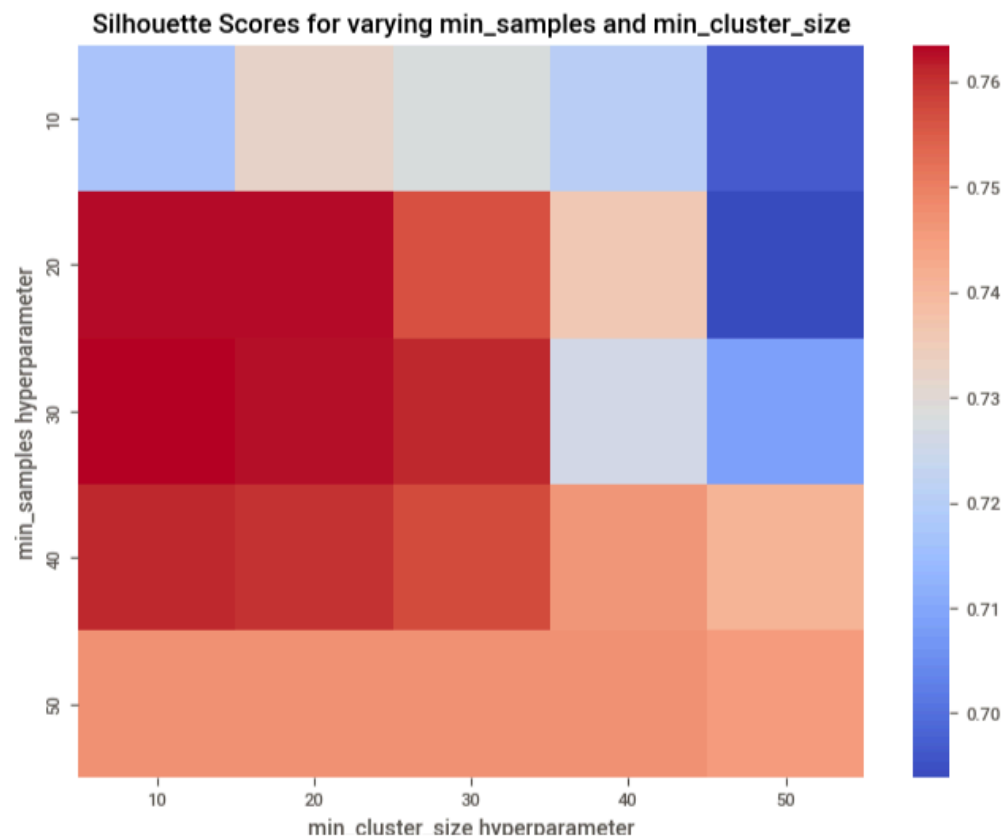
For DBSCAN we plotted heatmaps so we could compare both eps against min_samples:

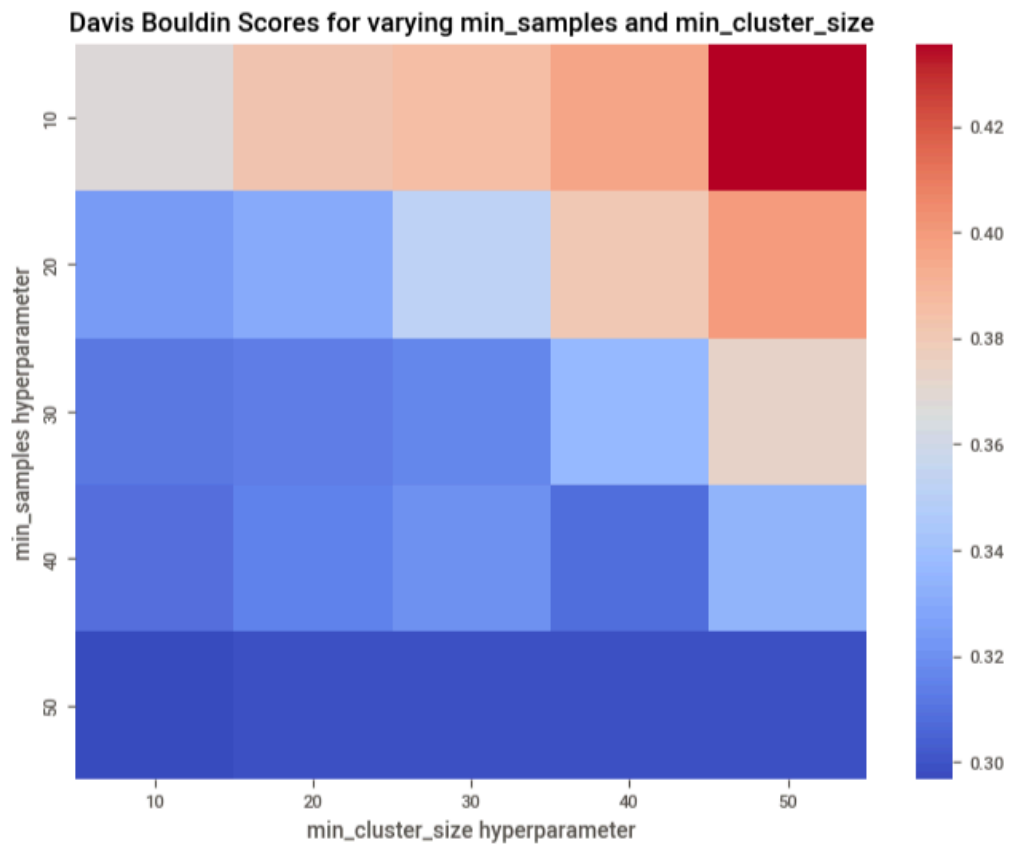


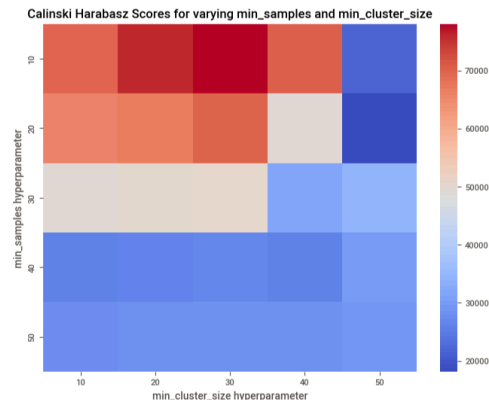




And lastly for HDBSCAN we plotted min_cluster_size against min_samples:








Our conclusions on the best hyperparameters that optimized the scores of each clustering algorithm are as follows:

Clustering Algorithm	Dataset	Hyperparameters	Inertia Score (Lower is better)	Silhouette Score (Higher is better)	Davies Bouldin Score (Lower is better)	Calinski Harabasz Score (Higher is better)
K-Means	Autoencoder	n_clusters=13	60,000	0.35	1.22	3750
K-Means	UMAP	n_clusters=15	200,000	0.49	0.78	9000
Hierarchical	Autoencoder	n_clusters=14	N/A	0.34	1.25	3750
Hierarchical	UMAP	n_clusters=14	N/A	0.58	0.63	9600
Birch	Autoencoder	n_clusters=17	N/A	0.36	1.20	3400
Birch	UMAP	n_clusters=19	N/A	0.56	0.64	11100
Meanshift	Autoencoder	n_clusters=18	N/A	0.38	1.22	3400
Meanshift	UMAP	n_clusters=15	N/A	0.58	0.60	9400
DBSCAN	Autoencoder	eps=0.1 and min_samples=50	N/A	0.80	0.30	70000

DBSCAN	UMAP	eps=0.1 and min_samples=50	N/A	0.97	0.04	800000
HDBSCAN	Autoencoder	min_cluster_size=30 and min_samples=50	N/A	0.47	0.75	3000
HDBSCAN	UMAP	min_cluster_size=20 and min_samples=10	N/A	0.76	0.32	75000

Using the best selected hyperparameters we calculated each score and used these scores to compare the algorithms against each other for our data:

	KMeans Silhouette Score: 0.4950469434261322 KMeans Davies Bouldin Score: 0.7784999366901298 KMeans Calinski Harabasz Score: 8979.162601032158
	Hierarchical Silhouette Score: 0.5846623182296753 Hierarchical Davies Bouldin Score: 0.6298731582925422 Hierarchical Calinski Harabasz Score: 9589.057842368
	Birch Silhouette Score: 0.5604621767997742 Birch Davies Bouldin Score: 0.6442845737411641 Birch Calinski Harabasz Score: 11177.971813794187
	Meanshift Silhouette Score: 0.5808581113815308 Meanshift Davies Bouldin Score: 0.6044274556705069 Meanshift Calinski Harabasz Score: 9351.738691225644
	DBScan Silhouette Score: 0.9702805280685425 DBScan Davies Bouldin Score: 0.041562971572436 DBScan Calinski Harabasz Score: 811388.2780834587
	HDBScan Silhouette Score: 0.7592822909355164 HDBScan Davies Bouldin Score: 0.32311908507926473 HDBScan Calinski Harabasz Score: 75394.43909491494

Initially we thought that this mean DBSCAN was by far the best performing algorithm, however, we realized these scores are only achieved because 90% of our data points are considered noise points with these hyperparameters. A model that cannot cluster 90% of the data is not useful to us.

It actually turned out that HDBSCAN was the best performing model with a silhouette and calinski harabasz score that is noticeably higher than all other algorithms, and a davies bouldin

score lower than all others. And with these hyperparameters only ~10% of data points are considered noise points.

Create Models with Chosen Hyperparameters

```
▶ kmeans_model, kmeans_labels = kmeans_clustering(umap_reduced_df, n_clusters=15)
hierarchical_model, hierarchical_labels = hierarchical_clustering(umap_reduced_df, n_clusters=14)
birch_model, birch_labels = birch_clustering(umap_reduced_df, n_clusters=19)
meanshift_model, meanshift_labels = meanshift_clustering(umap_reduced_df, n_clusters=15)
dbscan_model, dbscan_labels = dbscan_clustering(umap_reduced_df, min_samples=50, eps=0.1)
hdbscan_model, hdbscan_labels = hdbscan_clustering(umap_reduced_df, min_samples=20, min_cluster_size=10, prediction_data=True)
```

Results

In the end, it could be observed that HDBSCAN gave the best results with Birch being the next best. HDBSCAN found 206 clusters compared to the other algorithms, and out of 19,000 data points only 2,000 are noise points:

```
[ ] print(f"Num KMeans clusters: {len(np.unique(kmeans_labels))}")
    print(f"Num Hierarchical clusters: {len(np.unique(hierarchical_labels))}")
    print(f"Num Birch clusters: {len(np.unique(birch_labels))}")
    print(f"Num Meanshift clusters: {len(np.unique(meanshift_labels))}")
    print(f"Num DBSCAN clusters: {len(np.unique(dbscan_labels[valid_dbscan_labels]))}")
    print(f"Num HDBSCAN clusters: {len(np.unique(hdbscan_labels[valid_hdbscan_labels]))}")
```

```
➡ Num KMeans clusters: 15
   Num Hierarchical clusters: 14
   Num Birch clusters: 19
   Num Meanshift clusters: 15
   Num DBSCAN clusters: 17
   Num HDBSCAN clusters: 206
```

DBSCAN has significantly better internal metrics, but only because it found almost all data points to be noise.

HDBSCAN has the next best internal metrics and doesn't label so much noise.

```
[ ] print(f"Num DBSCAN noise points: {len(dbscan_labels) - len(dbscan_labels[valid_dbscan_labels])}")
    print(f"Num DBSCAN noise points: {len(hdbscan_labels) - len(hdbscan_labels[valid_hdbscan_labels])}")
    print(f"Total data points: {len(hdbscan_labels)}")
```

```
➡ Num DBSCAN noise points: 18100
   Num DBSCAN noise points: 2072
   Total data points: 19114
```

Without labeled data it is difficult to reason about the “correctness” of a clustering algorithm. So we printed the property types inside each cluster and can see that the clustering is working well. We considered attempting to compute a “purity” metric, but given the data is unlabeled, and it is not incorrect for a property type like CONDO and APARTMENT to be clustered together (number of bedrooms, bathrooms, etc, are also being considered). So at most we can say that it passes the eyeball test with each property types being clustered together for the most part:

```
(83, {'DUPLEX_TRIPLEX': 77})
(41, {'DUPLEX_TRIPLEX': 75})
(91, {'DUPLEX_TRIPLEX': 26})
(84, {'DUPLEX_TRIPLEX': 79})
(63, {'TOWNHOMES': 130})
(26, {'TOWNHOMES': 67})
(27, {'TOWNHOMES': 49})
(109, {'TOWNHOMES': 57})
(155, {'TOWNHOMES': 17})
(86, {'TOWNHOMES': 51})
(117, {'TOWNHOMES': 43})
(116, {'TOWNHOMES': 41})
(128, {'TOWNHOMES': 30})
(78, {'TOWNHOMES': 18})
(65, {'TOWNHOMES': 32})

(4, {'SINGLE_FAMILY': 673})
(171, {'SINGLE_FAMILY': 26})
(13, {'SINGLE_FAMILY': 443})
(0, {'SINGLE_FAMILY': 316})
(102, {'SINGLE_FAMILY': 39})
(60, {'SINGLE_FAMILY': 223})
(106, {'SINGLE_FAMILY': 48})
(3, {'SINGLE_FAMILY': 72})
(42, {'SINGLE_FAMILY': 235})
(103, {'SINGLE_FAMILY': 42})
(51, {'SINGLE_FAMILY': 108})
-
(8, {'CONDOS': 240})
(35, {'CONDOS': 37})
(12, {'CONDOS': 53})
(18, {'CONDOS': 179})
(19, {'CONDOS': 156})
(36, {'CONDOS': 119})
(179, {'CONDOS': 78})
(32, {'CONDOS': 156})
(68, {'CONDOS': 81})
(56, {'CONDOS': 7, 'APARTMENT': 52, 'TOWNHOMES': 2, 'OTHER': 1})
(112, {'CONDOS': 58, 'APARTMENT': 3})
(115, {'CONDOS': 61})
(1, {'CONDOS': 2, 'APARTMENT': 42})
(46, {'CONDOS': 206})
(49, {'CONDOS': 27})
(95, {'CONDOS': 31})
(45, {'CONDOS': 22})
(67, {'CONDOS': 57})
(118, {'CONDOS': 34})
(138, {'CONDOS': 2, 'APARTMENT': 33})
```

Lastly, we also performed k-fold cross validation to confirm that on different sliced data our algorithms were computing the same silhouette scores and that these results were not outliers. For example, here is the hierarchical clustering algorithm on 5 different k-folds of data:

```
from sklearn.model_selection import KFold
from sklearn.metrics import silhouette_score
from scipy.cluster.hierarchy import linkage, fcluster
import numpy as np
k_folds = 5
kf = KFold(n_splits=k_folds, shuffle=True, random_state=42)
silhouette_scores = []
for train_index, test_index in kf.split(umap_reduced_df):
    train_data, test_data = umap_reduced_df.iloc[train_index], umap_reduced_df.iloc[test_index]
    linkage_matrix = linkage(train_data, method='ward')
    n_clusters = 14
    train_labels = fcluster(linkage_matrix, t=n_clusters, criterion='maxclust')
    train_data_array = train_data.values
    centroids = np.array([train_data_array[train_labels == c].mean(axis=0) for c in np.unique(train_labels)])
    test_labels = np.argmax(np.linalg.norm(test_data.values[:, None] - centroids[None, :], axis=2), axis=1) + 1
    if len(np.unique(test_labels)) > 1:
        print(silhouette_score(test_data, test_labels))
        silhouette_scores.append(silhouette_score(test_data, test_labels))
    else:
        print("Skipping silhouette score computation due to only one cluster.")
mean_silhouette = np.mean(silhouette_scores) if silhouette_scores else None
print(f"Mean Silhouette Score: {mean_silhouette}" if mean_silhouette else "No valid silhouette score.")
```

0.56624466
0.5633252
0.589557
0.558923
0.56252974
Mean Silhouette Score: 0.5681159496307373

Conclusion

In this project, we successfully developed a rental recommendation web service that combines conversational AI with machine learning to improve the property search experience. Our system enables users to search for rental properties, refine their results through natural language interactions with a chatbot, and view recommendations for similar listings when exploring specific properties. The modular design of the application—comprising a React-based frontend, a FastAPI backend, a web scraper pipeline, and a recommendation model training pipeline—ensures scalability and maintainability.

Through this project, we gained insights into developing an end-to-end system for rental recommendations, managing diverse data sources. The integration of state-of-the-art tools like Apache Flink, Kafka, and AWS S3 has further solidified the robustness of the system.

We found that HDBSCAN provided the best scores of all our clustering algorithms, and it performed best on data that was dimensionally reduced using UMAP. With a silhouette score of 0.75, davies bouldin of 0.32, and calinski harabasz of 75000, it outperformed the other models.

Future Work

There are several opportunities to extend and enhance this system:

1. **Feedback-Driven Improvement:** Incorporating explicit user feedback to fine-tune the recommendation model further could improve its relevance and accuracy.
2. **Personalization:** Adding user profile-based personalization to offer tailored recommendations based on user history and preferences.
3. **Improve Chatbot NLP Capabilities:** We can make the recommendation system more user friendly by allowing the user to ask questions and get comprehensive answers regarding one type of property and their rent.
4. **Real-Time Recommendation Updates:** Enhancing the model to account for real-time trends in rental markets, such as price changes or new property listings.
5. **Enhanced Chatbot Features:** Expanding the chatbot's conversational capabilities to include more complex natural language understanding and support for additional languages.
6. **Mobile Application Integration:** Extending the system to a mobile application for wider accessibility.
7. **Expanded Data Sources:** Scraping additional rental platforms or integrating with APIs for richer data diversity.