# Arizona Housing Price Predictors

Kellie Halladay and Paulo Barrera

COMP 4531: Deep Learning; Model Design and Application

## Problem

The housing market has been a big topic of discussion for decades. Many individuals and families have concerns about their ability to afford a home in their lifetimes. Simultaneously, local real estate businesses struggle to compete against large private corporations that buy homes in bulk. Researchers have found plenty of trends in house pricing over the years. There are many factors that contribute to the price of a house. It is important to examine which attributes have the strongest correlation with price to get an understanding of which features are causing homes to be more expensive. The goal is to use real estate, school location, store location, population, and pricing times series trend datasets for the state of Arizona to create a model to predict the price of a house in Arizona based on location, size, number of beds/baths, garage, age of the house, distance to a school/grocery store, population, home value index, and date. This is a large problem that will likely involve multiple steps. First, we will need to determine the relationship between the various spatial features and the cost of the house. Then, we will need to find the trend in house prices over a long period of time, with the hope that the model we make will be able to predict the price of houses a few months in the future with reasonable accuracy. Adding this time series element will likely require a deeper model than linear regression, and a neural network could be very beneficial.

## Setup

Here is an overview of the variables used in the model, along with the type, range and encoding:

| Input/Output | Name | Description | Type | Categorical/ Continuous | Range | Encoding |
|---|---|---|---|---|---|---|
| Input | year | Year house was listed/sold | float32 | Continuous | 2023 - 2024 | None |
| Input | house_age | # of years since house was built | float32 | Continuous | -1 - 119 | None |
| Input | garage_ind | Garage indicator | float32 | Categorical | 0 - 1 | 0 - no garage<br>1 - garage |
| Input | price_per_sqft_num | Price per square feet of house | float32 | Continuous | 0 - 978 | None |
| Input | pop2024 | Population per city in 2024 | float32 | Continuous | 13 - 1676481 | None |
| Input | density | Population density per city in 2024 | float32 | Continuous | 8 - 4758 | None |
| Input | floor_space_num | Floor space in square feet | float32 | Continuous | 264 - 9461 | None |
| Input | min_school_dist | Distance to nearest school in km | float32 | Continuous | 0.00011 - 2.26181 | Smallest haversine distance between house and a school |
| Input | min_store_dist | Distance to nearest grocery store in km | float32 | Continuous | 0.00012 - 9.59067 | Smallest haversine distance between house and a store |

| | | | | | | |
|---|---|---|---|---|---|---|
| Input | beds_num | # of bedrooms | float32 | Continuous | 0 - 9 | None |
| Input | baths_num | # of bathrooms | float32 | Continuous | 0 - 9 | None |
| Input | city (multiple variables) | For each city variable, flag whether or not house is located in that city | float32 | Categorical | 0 - 1 | One Hot Encoded (1 for correct city, 0 otherwise) |
| Input | time | YYYY-MM-DD of record | datetime | Continuous | 2000-01-01 to 2024-03-31 | None |
| Input | HVI | Home value index | float32 | Continuous | 39586.76 - 1282502 | None |
| Input | months_since_start | Months since start of time data | int | Conscious | 0-74 | None |
| Output | price_num | Price of the house | float32 | Continuous | 100 - 5900000 | None |

## Problem Exploration

Since we are ultimately trying to train the data we use to make a successful and accurate model, it is crucial to explore the data we use, handle missing values, examine each variable's distribution, find the correlation between the variables, format each one to align with our needs, and try fitting some baseline models (linear regression, random forest, etc).

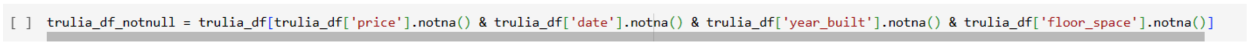
For the spatial data, we will examine a few data files. First, we will utilize a 'trulia_data.csv' file we created containing information scraped from Arizona listings found on https://www.trulia.com from July 2023 to April 2024. Note that the method used for scraping the data involved collecting information from listings for different price ranges, starting with prices less than $50K, then prices in a range from $50K-$100K, then $100K-$150K, and so on, up until the final range of greater than $1 million.  Each dataset was then combined into one.  So there is a large range of prices included in our Trulia dataset, and the listings chosen from each group were randomly generated from Trulia.  So the data should not be biased.

Next, we use a 'Arizona_stores_latlong.csv' dataset with information on all grocery store locations in Arizona scraped from http://www.supermarketpage.com/state/AZ/ with latitude and longitude coordinates found from https://www.gps-coordinates.net/gps-coordinates-converter

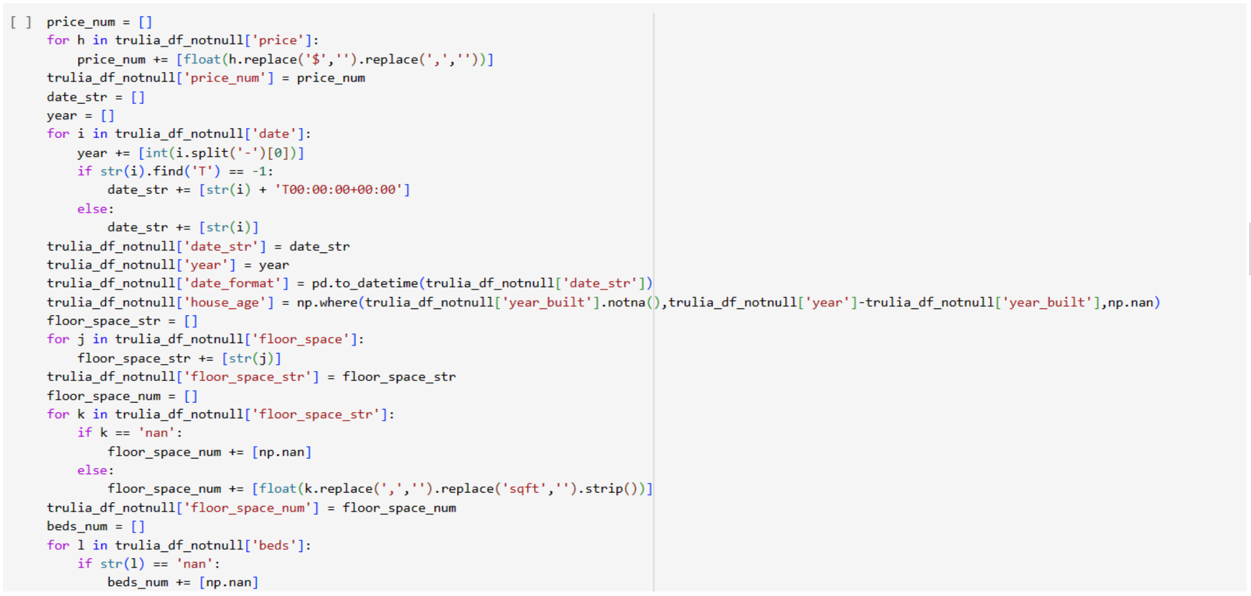
Then, we also utilize a 'Schools.xlsx' dataset with locations of all the schools in Arizona, obtained from https://azgeo-data-hub-agic.hub.arcgis.com/datasets/29abc422fd0541f2b3bbe21d1b16c5f6/explore.

Finally, we will utilize a 'us-cities-table-for-arizona.csv' dataset containing information on the population per city, obtained from https://worldpopulationreview.com/states/cities/arizona.

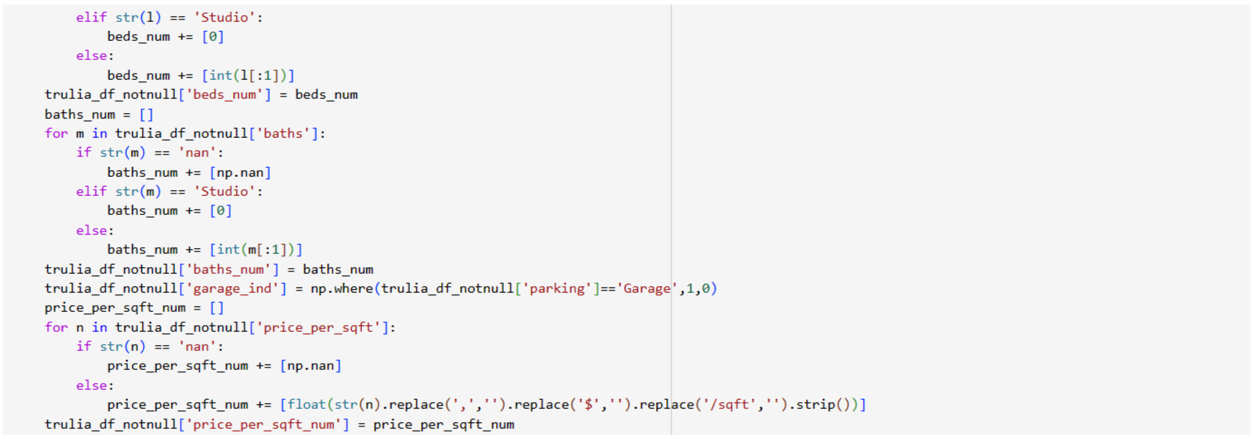
The trulia dataset can now be transformed in order to be used for modeling.  This includes removing rows with null values in some essential fields (price, year built, floorspace), converting many variables to numeric, and computing some new variables (house age):



```
[ ] trulia_df_notnull = trulia_df[trulia_df['price'].notna() & trulia_df['date'].notna() & trulia_df['year_built'].notna() & trulia_df['floor_space'].notna()]
```



```
[ ] price_num = []
    for h in trulia_df_notnull['price']:
        price_num += [float(h.replace('$','').replace(',',''))]
    trulia_df_notnull['price_num'] = price_num
    date_str = []
    year = []
    for i in trulia_df_notnull['date']:
        year += [int(i.split('-')[0])]
        if str(i).find('T') == -1:
            date_str += [str(i) + 'T00:00:00+00:00']
        else:
            date_str += [str(i)]
    trulia_df_notnull['date_str'] = date_str
    trulia_df_notnull['year'] = year
    trulia_df_notnull['date_format'] = pd.to_datetime(trulia_df_notnull['date_str'])
    trulia_df_notnull['house_age'] = np.where(trulia_df_notnull['year_built'].notna(),trulia_df_notnull['year']-trulia_df_notnull['year_built'],np.nan)
    floor_space_str = []
    for j in trulia_df_notnull['floor_space']:
        floor_space_str += [str(j)]
    trulia_df_notnull['floor_space_str'] = floor_space_str
    floor_space_num = []
    for k in trulia_df_notnull['floor_space_str']:
        if k == 'nan':
            floor_space_num += [np.nan]
        else:
            floor_space_num += [float(k.replace(',','').replace('sqft','').strip())]
    trulia_df_notnull['floor_space_num'] = floor_space_num
    beds_num = []
    for l in trulia_df_notnull['beds']:
        if str(l) == 'nan':
            beds_num += [np.nan]
```



```
        elif str(l) == 'Studio':
            beds_num += [0]
        else:
            beds_num += [int(l[:1])]
    trulia_df_notnull['beds_num'] = beds_num
    baths_num = []
    for m in trulia_df_notnull['baths']:
        if str(m) == 'nan':
            baths_num += [np.nan]
        elif str(m) == 'Studio':
            baths_num += [0]
        else:
            baths_num += [int(m[:1])]
    trulia_df_notnull['baths_num'] = baths_num
    trulia_df_notnull['garage_ind'] = np.where(trulia_df_notnull['parking']=='Garage',1,0)
    price_per_sqft_num = []
    for n in trulia_df_notnull['price_per_sqft']:
        if str(n) == 'nan':
            price_per_sqft_num += [np.nan]
        else:
            price_per_sqft_num += [float(str(n).replace(',','').replace('$','').replace('/sqft','').strip())]
    trulia_df_notnull['price_per_sqft_num'] = price_per_sqft_num
```

This transformed data will now be merged with the population data by the city, to incorporate additional population variables into the model.

Next, we need a way to find the distance between two coordinates, in order to determine the minimum distance between each house and a school/grocery store. "The haversine formula is a very accurate way of computing distances between two points on the surface of a sphere using the latitude and longitude of the two points. The haversine formula is a re-formulation of the spherical law of cosines, but the formulation in terms of haversines is more useful for small angles and distances" (Kettle). The haversine formula involves the following three steps, where

$(\phi_A, \lambda_A)$ and $(\phi_B, \lambda_B)$ are the coordinates of two points, and the final equation $d$ yields the distance between these points:

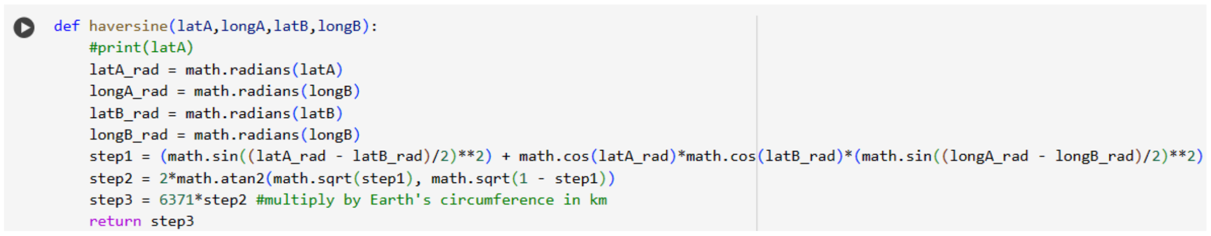
$$a = sin^2\left(\frac{\phi_B - \phi_A}{2}\right) + cos\left(\phi_A\right)cos\left(\phi_B\right)sin^2\left(\frac{\lambda_B - \lambda_A}{2}\right)$$

$$c = 2tan^{-1}\left(\sqrt{\frac{a}{1-a}}\right)$$

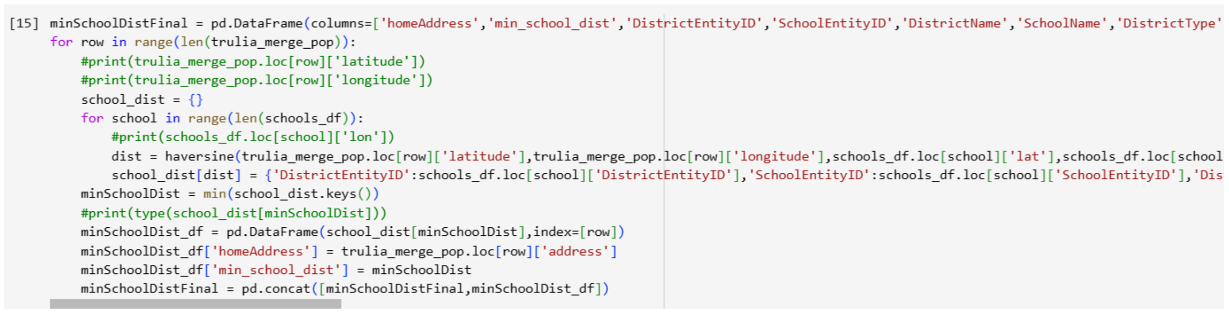
$$d = 6371c$$

(Kettle)

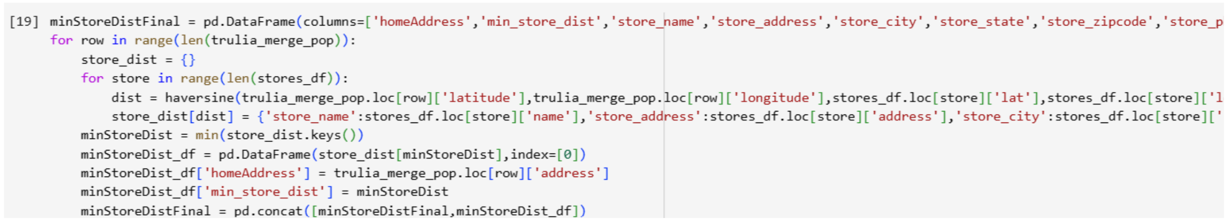
Below is the code for the haversine formula, as well as the code that applies this formula to find the distance of every school to each house, and computes the minimum of the distances per house. The same logic is applied to find minimum distances to stores. Then these minimum distance data frames are merged with the main dataframe to create one dataframe with all the spatial data fields needed:



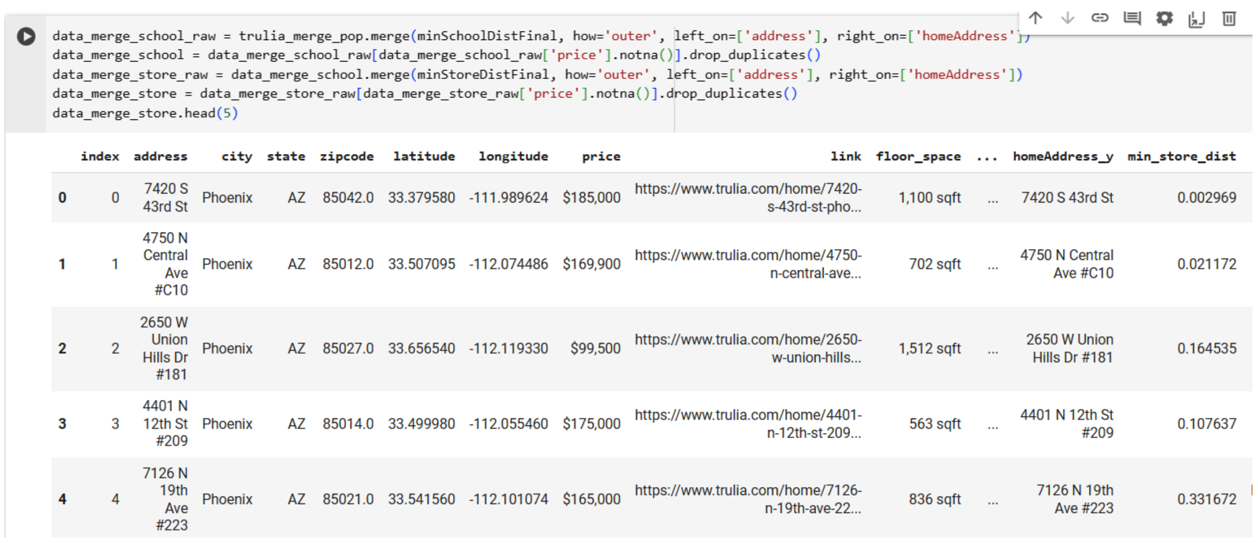
```
def haversine(latA,longA,latB,longB):
    #print(latA)
    latA_rad = math.radians(latA)
    longA_rad = math.radians(longB)
    latB_rad = math.radians(latB)
    longB_rad = math.radians(longB)
    step1 = (math.sin((latA_rad - latB_rad)/2)**2) + math.cos(latA_rad)*math.cos(latB_rad)*(math.sin((longA_rad - longB_rad)/2)**2)
    step2 = 2*math.atan2(math.sqrt(step1), math.sqrt(1 - step1))
    step3 = 6371*step2 #multiply by Earth's circumference in km
    return step3
```



```
[15] minSchoolDistFinal = pd.DataFrame(columns=['homeAddress','min_school_dist','DistrictEntityID','SchoolEntityID','DistrictName','SchoolName','DistrictType'
     for row in range(len(trulia_merge_pop)):
         #print(trulia_merge_pop.loc[row]['latitude'])
         #print(trulia_merge_pop.loc[row]['longitude'])
         school_dist = {}
         for school in range(len(schools_df)):
             #print(schools_df.loc[school]['lon'])
             dist = haversine(trulia_merge_pop.loc[row]['latitude'],trulia_merge_pop.loc[row]['longitude'],schools_df.loc[school]['lat'],schools_df.loc[school
             school_dist[dist] = {'DistrictEntityID':schools_df.loc[school]['DistrictEntityID'],'SchoolEntityID':schools_df.loc[school]['SchoolEntityID'],'Dis
         minSchoolDist = min(school_dist.keys())
         #print(type(school_dist[minSchoolDist]))
         minSchoolDist_df = pd.DataFrame(school_dist[minSchoolDist],index=[row])
         minSchoolDist_df['homeAddress'] = trulia_merge_pop.loc[row]['address']
         minSchoolDist_df['min_school_dist'] = minSchoolDist
         minSchoolDistFinal = pd.concat([minSchoolDistFinal,minSchoolDist_df])
```



```
[19] minStoreDistFinal = pd.DataFrame(columns=['homeAddress','min_store_dist','store_name','store_address','store_city','store_state','store_zipcode','store_p
     for row in range(len(trulia_merge_pop)):
         store_dist = {}
         for store in range(len(stores_df)):
             dist = haversine(trulia_merge_pop.loc[row]['latitude'],trulia_merge_pop.loc[row]['longitude'],stores_df.loc[store]['lat'],stores_df.loc[store]['l
             store_dist[dist] = {'store_name':stores_df.loc[store]['name'],'store_address':stores_df.loc[store]['address'],'store_city':stores_df.loc[store]['
         minStoreDist = min(store_dist.keys())
         minStoreDist_df = pd.DataFrame(store_dist[minStoreDist],index=[0])
         minStoreDist_df['homeAddress'] = trulia_merge_pop.loc[row]['address']
         minStoreDist_df['min_store_dist'] = minStoreDist
         minStoreDistFinal = pd.concat([minStoreDistFinal,minStoreDist_df])
```

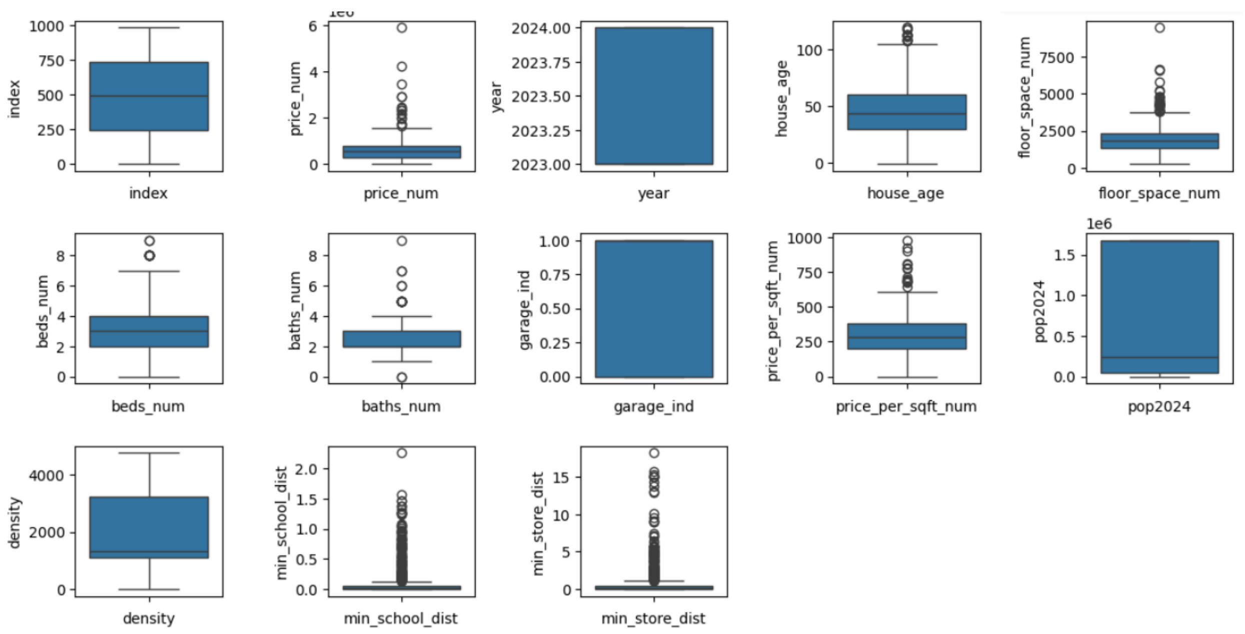


```
data_merge_school_raw = trulia_merge_pop.merge(minSchoolDistFinal, how='outer', left_on=['address'], right_on=['homeAddress'])
data_merge_school = data_merge_school_raw[data_merge_school_raw['price'].notna()].drop_duplicates()
data_merge_store_raw = data_merge_school.merge(minStoreDistFinal, how='outer', left_on=['address'], right_on=['homeAddress'])
data_merge_store = data_merge_store_raw[data_merge_store_raw['price'].notna()].drop_duplicates()
data_merge_store.head(5)
```

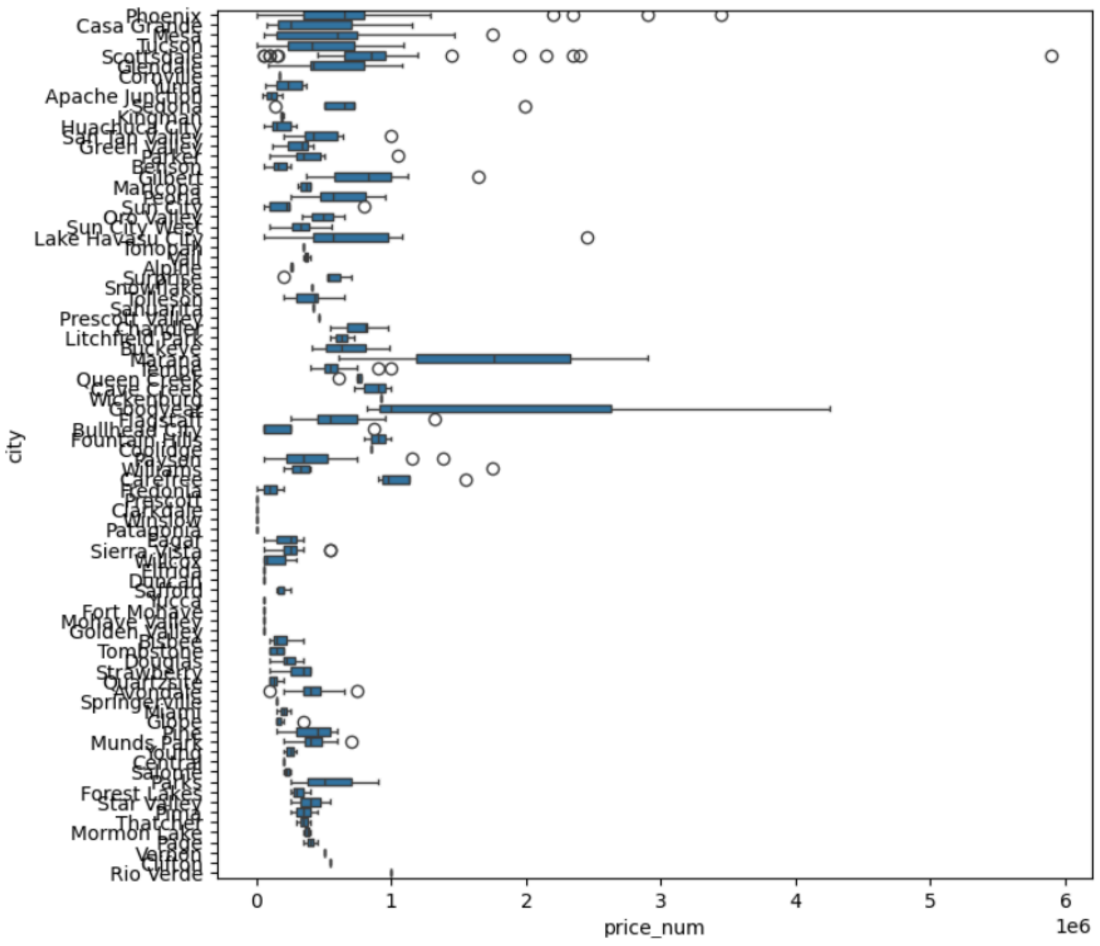
| | index | address | city | state | zipcode | latitude | longitude | price | link | floor_space | ... | homeAddress_y | min_store_dist |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 7420 S 43rd St | Phoenix | AZ | 85042.0 | 33.379580 | -111.989624 | $185,000 | https://www.trulia.com/home/7420-s-43rd-st-pho... | 1,100 sqft | ... | 7420 S 43rd St | 0.002969 |
| 1 | 1 | 4750 N Central Ave #C10 | Phoenix | AZ | 85012.0 | 33.507095 | -112.074486 | $169,900 | https://www.trulia.com/home/4750-n-central-ave... | 702 sqft | ... | 4750 N Central Ave #C10 | 0.021172 |
| 2 | 2 | 2650 W Union Hills Dr #181 | Phoenix | AZ | 85027.0 | 33.656540 | -112.119330 | $99,500 | https://www.trulia.com/home/2650-w-union-hills... | 1,512 sqft | ... | 2650 W Union Hills Dr #181 | 0.164535 |
| 3 | 3 | 4401 N 12th St #209 | Phoenix | AZ | 85014.0 | 33.499980 | -112.055460 | $175,000 | https://www.trulia.com/home/4401-n-12th-st-209... | 563 sqft | ... | 4401 N 12th St #209 | 0.107637 |
| 4 | 4 | 7126 N 19th Ave #223 | Phoenix | AZ | 85021.0 | 33.541560 | -112.101074 | $165,000 | https://www.trulia.com/home/7126-n-19th-ave-22... | 836 sqft | ... | 7126 N 19th Ave #223 | 0.331672 |

Now that all the spatial data has been put into one dataframe, it is essential to explore the dataset, examine the distribution of the variables, handle outliers and missing data, etc. First, we will remove any extra fields that we don't need and keep only the variables of interest: price_num, city, year, house_age, floor_space_num, beds_num, baths_num, garage_ind, price_per_sqft_num, pop2024, density, min_school_dist, and min_store_dist. Next, when taking summary statistics of the variables, we found that the price_num, price_per_sqft_num, and floor_space have a large gap between the maximum value and the 75th percentile value. So there are some outliers that may skew the mean values of our variables and could impact our models. To account for this, we filtered out the extreme values.

Here are the distributions of our variables represented with boxplots.



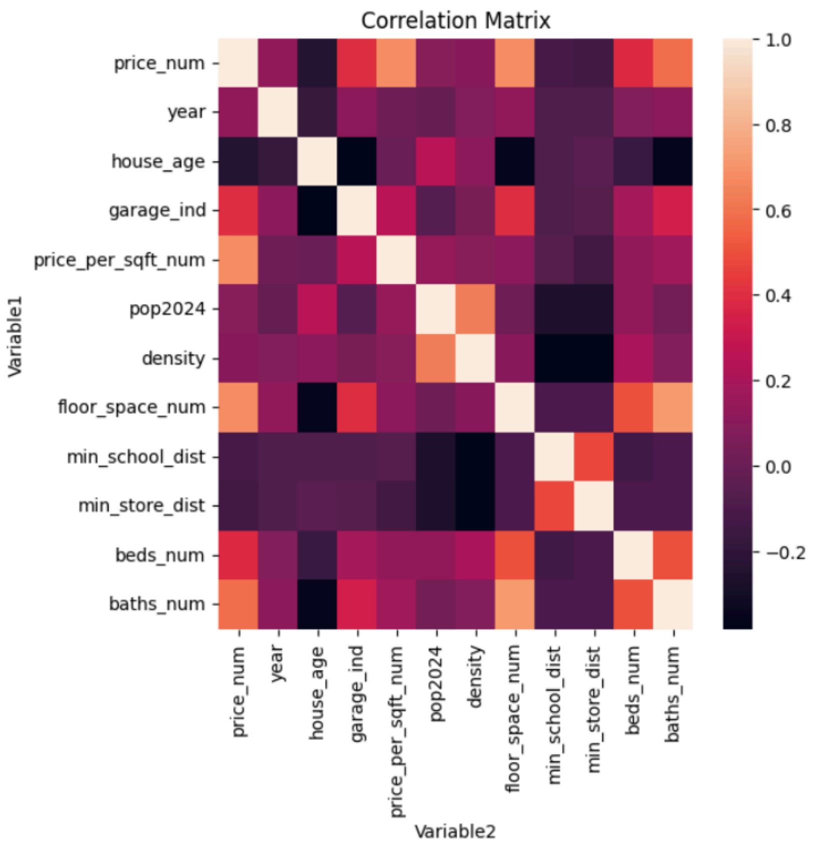
Next, here is a visual of the price spread per city:



It appears that the city has an effect on housing prices. For example, Marana and Goodyear tend to have higher house costs than other cities.

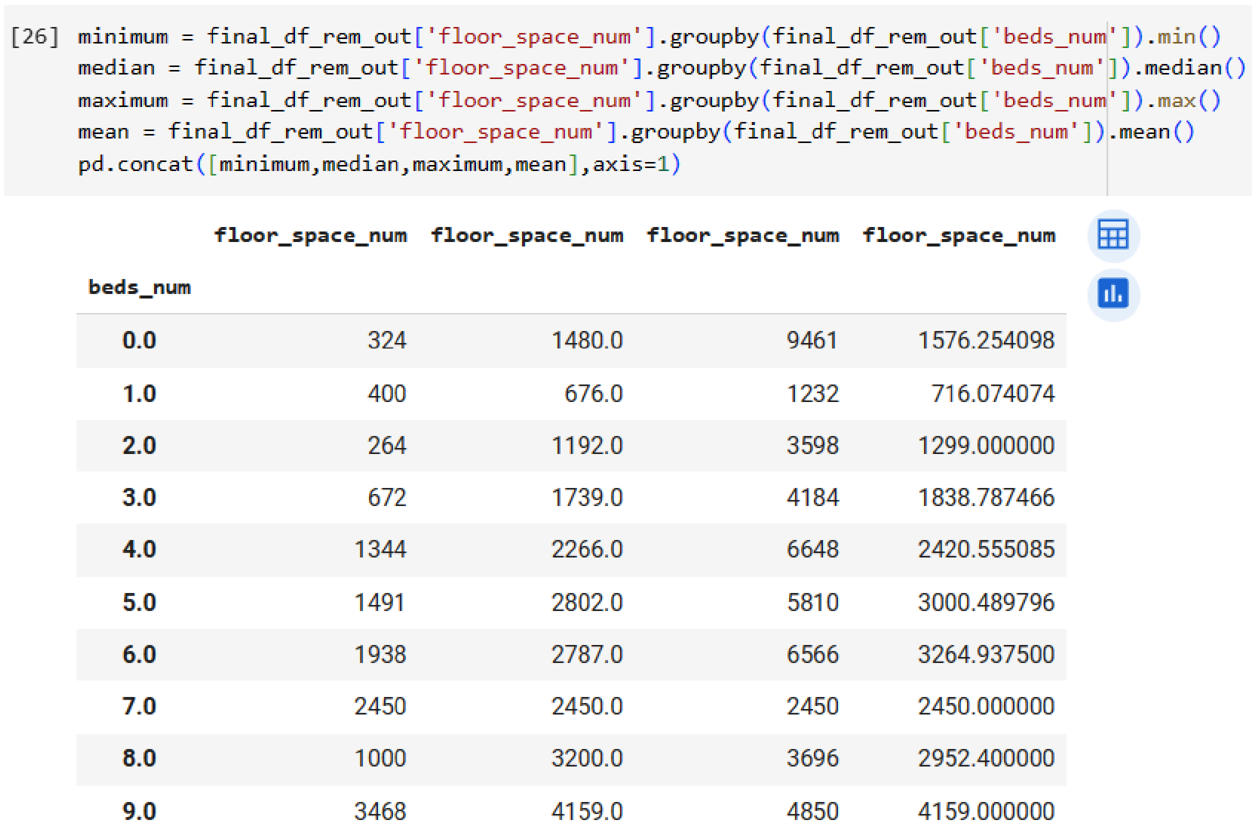
Now, we will observe the correlation between all the variables with a correlation matrix and a heatmap.

| | v1 | v2 | correlation |
|---|---|---|---|
| 0 | price_num | price_num | 1.000000 |
| 7 | price_num | price_per_sqft_num | 0.685268 |
| 3 | price_num | floor_space_num | 0.672421 |
| 5 | price_num | baths_num | 0.579403 |
| 6 | price_num | garage_ind | 0.397230 |
| 4 | price_num | beds_num | 0.384528 |
| 2 | price_num | house_age | -0.241300 |
| 11 | price_num | min_store_dist | -0.138788 |
| 1 | price_num | year | 0.125881 |
| 10 | price_num | min_school_dist | -0.111873 |
| 9 | price_num | density | 0.100906 |
| 8 | price_num | pop2024 | 0.091334 |



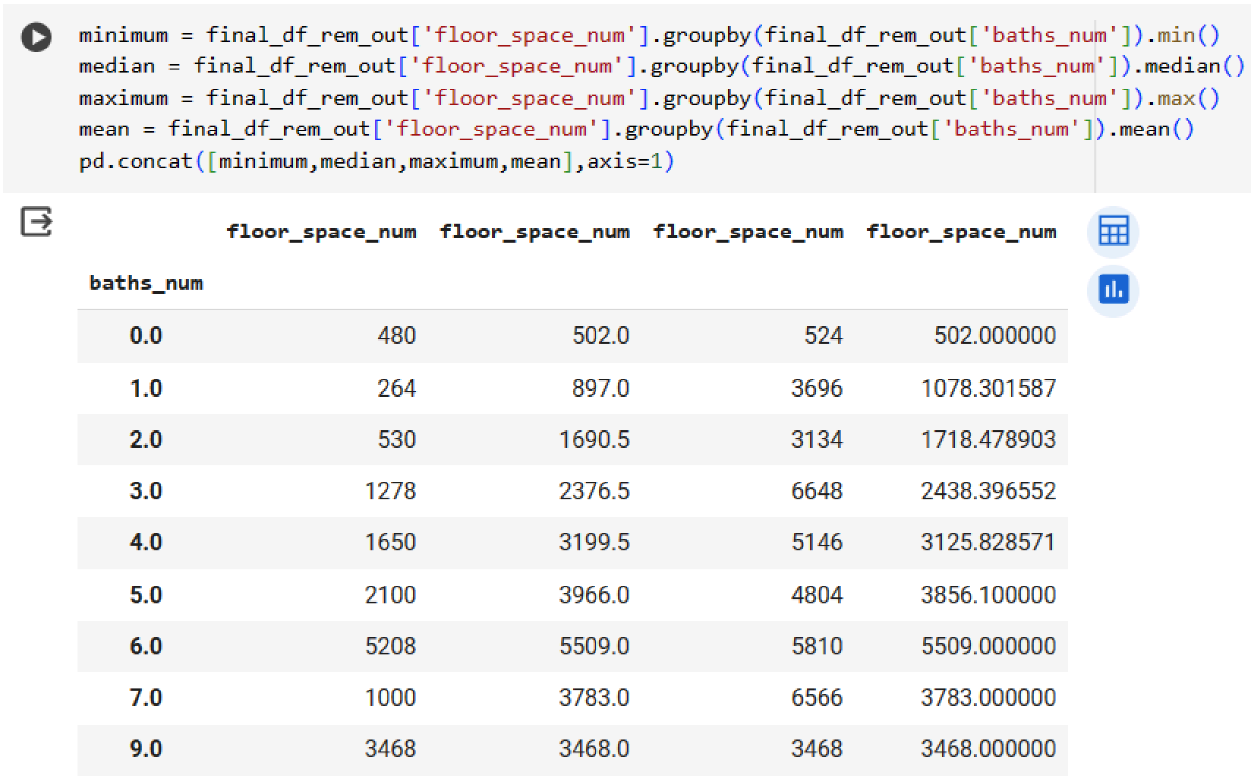
It appears that the variables most strongly correlated to the price are price_per_sqft_num, floor_space_num, baths_num, and garage_ind.  Also, another observation is that the floor_space_num, beds_num, and baths_num are strongly correlated with each other.

Next, we took counts of the missing values for each variable in our dataset and found that there are 6 missing beds_num and 57 baths_num values.  To account for this, we will need to impute new values where data is missing.  Since these fields are strongly correlated to the floor_space_num variable, we will examine the floor_space_num value ranges for each distinct beds value and baths value that's not missing.



```
[26] minimum = final_df_rem_out['floor_space_num'].groupby(final_df_rem_out['beds_num']).min()
     median = final_df_rem_out['floor_space_num'].groupby(final_df_rem_out['beds_num']).median()
     maximum = final_df_rem_out['floor_space_num'].groupby(final_df_rem_out['beds_num']).max()
     mean = final_df_rem_out['floor_space_num'].groupby(final_df_rem_out['beds_num']).mean()
     pd.concat([minimum,median,maximum,mean],axis=1)
```

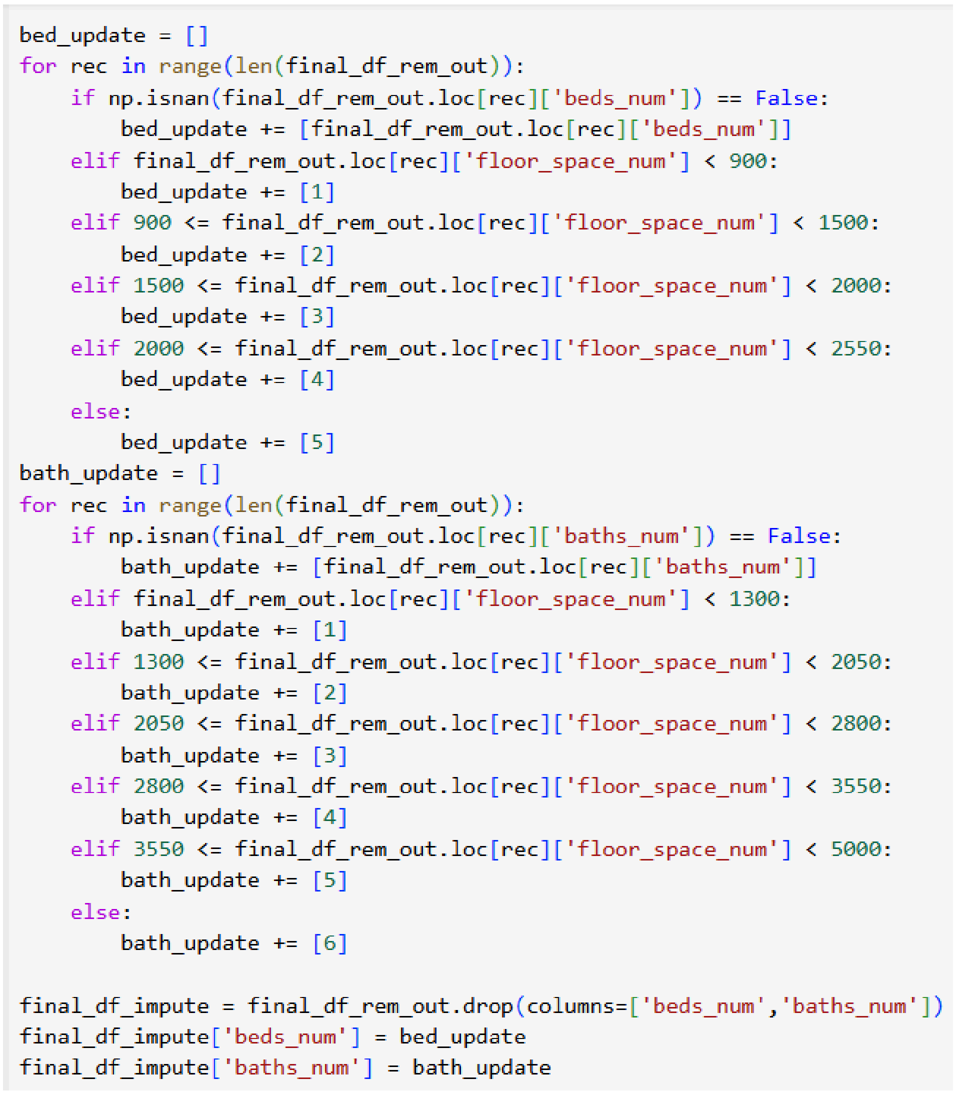
| | floor_space_num | floor_space_num | floor_space_num | floor_space_num |
|---|---|---|---|---|
| beds_num | | | | |
| 0.0 | 324 | 1480.0 | 9461 | 1576.254098 |
| 1.0 | 400 | 676.0 | 1232 | 716.074074 |
| 2.0 | 264 | 1192.0 | 3598 | 1299.000000 |
| 3.0 | 672 | 1739.0 | 4184 | 1838.787466 |
| 4.0 | 1344 | 2266.0 | 6648 | 2420.555085 |
| 5.0 | 1491 | 2802.0 | 5810 | 3000.489796 |
| 6.0 | 1938 | 2787.0 | 6566 | 3264.937500 |
| 7.0 | 2450 | 2450.0 | 2450 | 2450.000000 |
| 8.0 | 1000 | 3200.0 | 3696 | 2952.400000 |
| 9.0 | 3468 | 4159.0 | 4850 | 4159.000000 |



```
minimum = final_df_rem_out['floor_space_num'].groupby(final_df_rem_out['baths_num']).min()
median = final_df_rem_out['floor_space_num'].groupby(final_df_rem_out['baths_num']).median()
maximum = final_df_rem_out['floor_space_num'].groupby(final_df_rem_out['baths_num']).max()
mean = final_df_rem_out['floor_space_num'].groupby(final_df_rem_out['baths_num']).mean()
pd.concat([minimum,median,maximum,mean],axis=1)
```

| | floor_space_num | floor_space_num | floor_space_num | floor_space_num |
|---|---|---|---|---|
| **baths_num** | | | | |
| **0.0** | 480 | 502.0 | 524 | 502.000000 |
| **1.0** | 264 | 897.0 | 3696 | 1078.301587 |
| **2.0** | 530 | 1690.5 | 3134 | 1718.478903 |
| **3.0** | 1278 | 2376.5 | 6648 | 2438.396552 |
| **4.0** | 1650 | 3199.5 | 5146 | 3125.828571 |
| **5.0** | 2100 | 3966.0 | 4804 | 3856.100000 |
| **6.0** | 5208 | 5509.0 | 5810 | 5509.000000 |
| **7.0** | 1000 | 3783.0 | 6566 | 3783.000000 |
| **9.0** | 3468 | 3468.0 | 3468 | 3468.000000 |

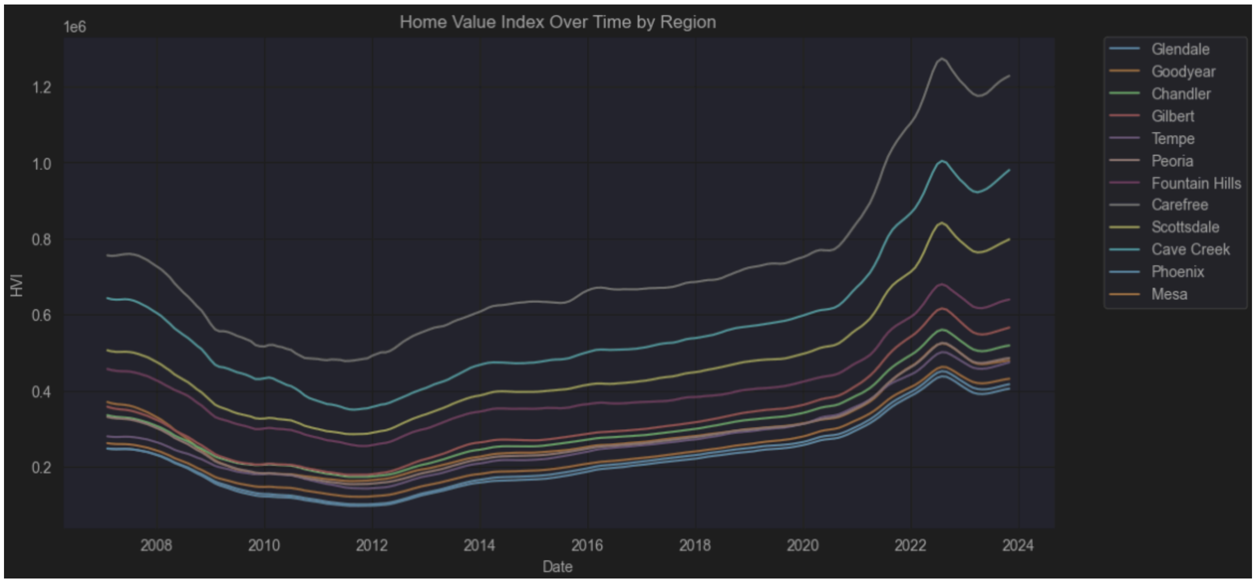
Notice that the ranges seemed to be ordered in a way such that as the number of beds/baths increases, the floor space value ranges tend to increase for the most part. So in cases where the beds or baths are missing, we will choose a value for those fields based on the floor space (i.e. if the floorspace is within a given range, the number of beds will be listed as 0, if the floorspace is within a different range, the beds will be 1, etc). We will determine the floor space range cutoff points by finding the values about halfway between the medians of the floor spaces for two consecutive beds/baths values.



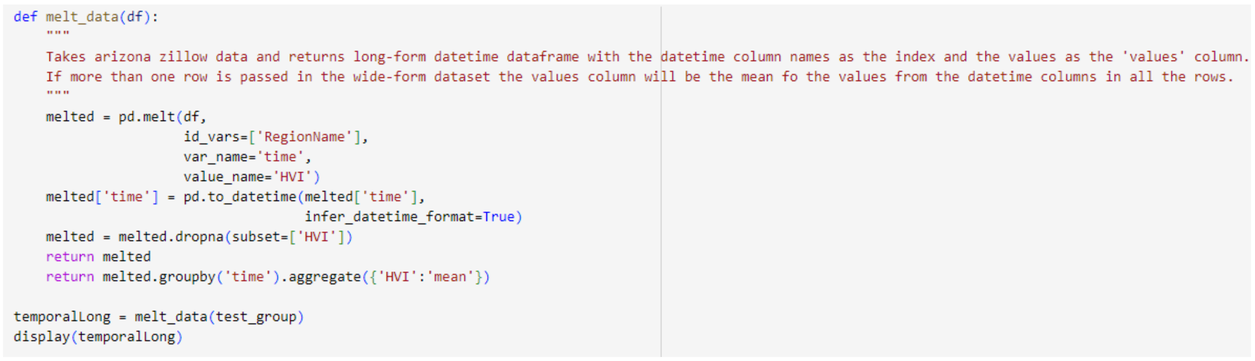
```python
bed_update = []
for rec in range(len(final_df_rem_out)):
    if np.isnan(final_df_rem_out.loc[rec]['beds_num']) == False:
        bed_update += [final_df_rem_out.loc[rec]['beds_num']]
    elif final_df_rem_out.loc[rec]['floor_space_num'] < 900:
        bed_update += [1]
    elif 900 <= final_df_rem_out.loc[rec]['floor_space_num'] < 1500:
        bed_update += [2]
    elif 1500 <= final_df_rem_out.loc[rec]['floor_space_num'] < 2000:
        bed_update += [3]
    elif 2000 <= final_df_rem_out.loc[rec]['floor_space_num'] < 2550:
        bed_update += [4]
    else:
        bed_update += [5]
bath_update = []
for rec in range(len(final_df_rem_out)):
    if np.isnan(final_df_rem_out.loc[rec]['baths_num']) == False:
        bath_update += [final_df_rem_out.loc[rec]['baths_num']]
    elif final_df_rem_out.loc[rec]['floor_space_num'] < 1300:
        bath_update += [1]
    elif 1300 <= final_df_rem_out.loc[rec]['floor_space_num'] < 2050:
        bath_update += [2]
    elif 2050 <= final_df_rem_out.loc[rec]['floor_space_num'] < 2800:
        bath_update += [3]
    elif 2800 <= final_df_rem_out.loc[rec]['floor_space_num'] < 3550:
        bath_update += [4]
    elif 3550 <= final_df_rem_out.loc[rec]['floor_space_num'] < 5000:
        bath_update += [5]
    else:
        bath_update += [6]

final_df_impute = final_df_rem_out.drop(columns=['beds_num','baths_num'])
final_df_impute['beds_num'] = bed_update
final_df_impute['baths_num'] = bath_update
```

Now that the above logic is applied, there are no more remaining missing values in the dataset.

For the temporal data that was retrieved from zillow, https://www.zillow.com/research/data/, the zillow data came in the form of region, date and home value index. For the regions we are looking at this data is complete, those being cities whose HVI's are actually recorded.

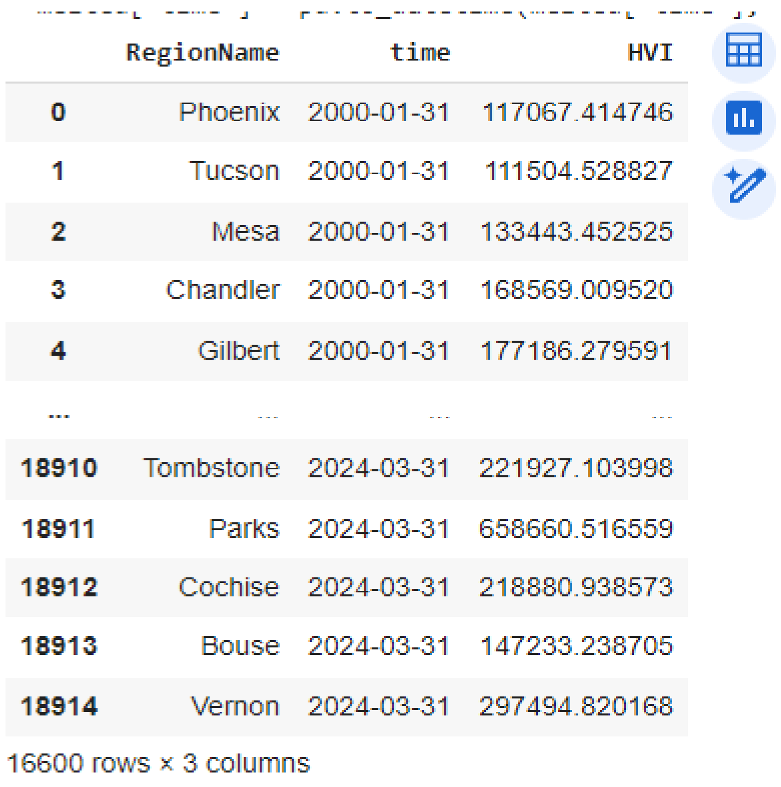


There was a little extra data added and the columns were the date so I created a function to melt the data into the form we wanted.



```
def melt_data(df):
    """
    Takes arizona zillow data and returns long-form datetime dataframe with the datetime column names as the index and the values as the 'values' column.
    If more than one row is passed in the wide-form dataset the values column will be the mean fo the values from the datetime columns in all the rows.
    """
    melted = pd.melt(df,
                     id_vars=['RegionName'],
                     var_name='time',
                     value_name='HVI')
    melted['time'] = pd.to_datetime(melted['time'],
                                    infer_datetime_format=True)
    melted = melted.dropna(subset=['HVI'])
    return melted
    return melted.groupby('time').aggregate({'HVI':'mean'})

temporalLong = melt_data(test_group)
display(temporalLong)
```

This puts our data into a much nicer form that will make creating the training and testing data much easier.

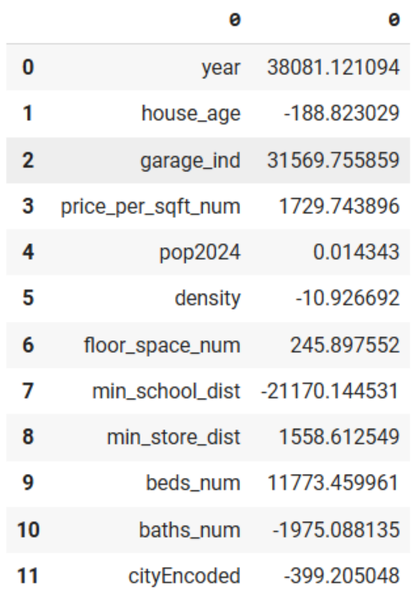


| | RegionName | time | HVI |
|---|---|---|---|
| 0 | Phoenix | 2000-01-31 | 117067.414746 |
| 1 | Tucson | 2000-01-31 | 111504.528827 |
| 2 | Mesa | 2000-01-31 | 133443.452525 |
| 3 | Chandler | 2000-01-31 | 168569.009520 |
| 4 | Gilbert | 2000-01-31 | 177186.279591 |
| ... | ... | ... | ... |
| 18910 | Tombstone | 2024-03-31 | 221927.103998 |
| 18911 | Parks | 2024-03-31 | 658660.516559 |
| 18912 | Cochise | 2024-03-31 | 218880.938573 |
| 18913 | Bouse | 2024-03-31 | 147233.238705 |
| 18914 | Vernon | 2024-03-31 | 297494.820168 |

16600 rows × 3 columns

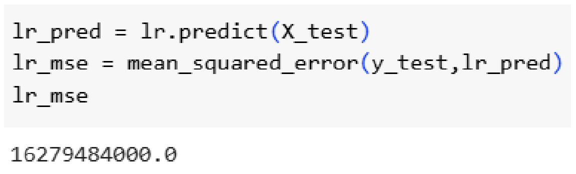
From here we needed to find a way to encode the data and we decided that one hot encoding would make our data too messy so we ended up using the LabelEncoder method from the sklearn.preprocessing package and applied that to both the spatial and temporal data. We then are using data from 2018-2023 as the feature set and will be predicting the values for the last 6 months of the data set (10/1/2023-3/1/2024). The targets and features will be tuned into numpy arrays and reshaped into the correct shape so that the number of feature sequences will be the same as the number of target values. We also added a continuous month column that would add a new column that labeled each row with the month from the first time entry to the last time entry.

Now that the data has been cleaned, formatted, and explored, we can start developing models. Linear regression is usually a good place to start, as it is one of the more basic model types. First, we will divide the spatial data into a target set, which has the price_num variable, and the feature set, which has every other variable, and split these sets into training sets, validation sets, and test sets (60 - 20 - 20 split). We will utilize scikit-learn's linear regression module to fit the training data to a linear model, and examine the coefficients produced from the model.



|  | 0 | 0 |
|---|---|---|
| **0** | year | 38081.121094 |
| **1** | house_age | -188.823029 |
| **2** | garage_ind | 31569.755859 |
| **3** | price_per_sqft_num | 1729.743896 |
| **4** | pop2024 | 0.014343 |
| **5** | density | -10.926692 |
| **6** | floor_space_num | 245.897552 |
| **7** | min_school_dist | -21170.144531 |
| **8** | min_store_dist | 1558.612549 |
| **9** | beds_num | 11773.459961 |
| **10** | baths_num | -1975.088135 |
| **11** | cityEncoded | -399.205048 |

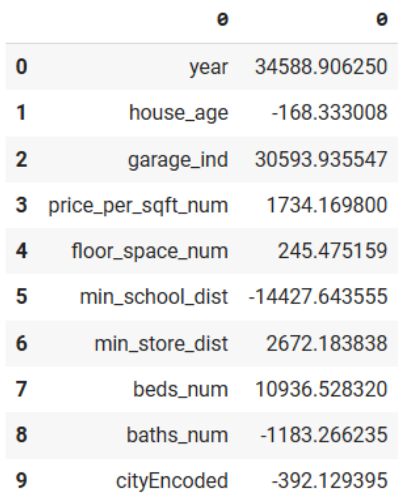
Notice that there are very small coefficients for the pop2024 (0.014343) and density variables (-10.926692). Now that the model has been created, the test data can be run through it, and the output will be compared to the actual values to compute the mean squared error:



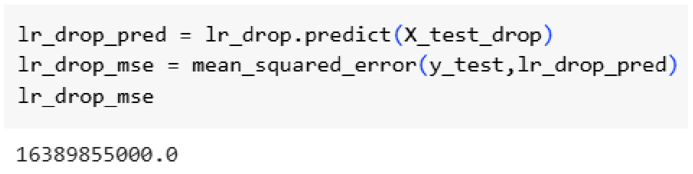
```
lr_pred = lr.predict(X_test)
lr_mse = mean_squared_error(y_test,lr_pred)
lr_mse
```

```
16279484000.0
```

We have a very high mean squared error value here, which indicates that this model is not very accurate. Next, we will try dropping the pop2024 and density variables from the feature sets, given that they have small coefficients and are not making a big impact on the outcome. We will create a new linear model without these variables and apply it to our test set again:



|   | 0 | 0 |
|---|---|---|
| 0 | year | 34588.906250 |
| 1 | house_age | -168.333008 |
| 2 | garage_ind | 30593.935547 |
| 3 | price_per_sqft_num | 1734.169800 |
| 4 | floor_space_num | 245.475159 |
| 5 | min_school_dist | -14427.643555 |
| 6 | min_store_dist | 2672.183838 |
| 7 | beds_num | 10936.528320 |
| 8 | baths_num | -1183.266235 |
| 9 | cityEncoded | -392.129395 |



```
lr_drop_pred = lr_drop.predict(X_test_drop)
lr_drop_mse = mean_squared_error(y_test,lr_drop_pred)
lr_drop_mse
```

```
16389855000.0
```

As you will see, there is a slightly smaller mean squared error value, but overall, dropping those two variables did not make an impactful difference. It appears that linear regression is not the best choice for this problem.

Now, we will try random forest regression models. This is an ensemble model type that creates multiple decision trees (models generated by repeatedly splitting the data into subsets based on the values of the features) and the output is selected based on the mean prediction of all the trees. We will employ scikit-learn's random forest regression module and utilize the cross_val_score functionality to split the data into multiple training and test sets, and for each training and test set, train the model on the training set and make predictions on the test set, and output the mean squared error value for each run. This will be done for both the full feature set, and the set with the pop2024 and density variables dropped.

```
#Random Forest Regressor
rfr = RandomForestRegressor(random_state=42)
rfr.fit(X_train,y_train)
rfr_cv_scores = cross_val_score(rfr, X_train, y_train, scoring='neg_mean_squared_error', cv=10)
rfr_mse_scores = -rfr_cv_scores
rfr_mse_scores
```

```
array([1.08120592e+09, 8.87954741e+08, 6.50926379e+09, 6.40951556e+10,
       1.52885854e+09, 1.03660222e+09, 1.14952625e+10, 3.18019507e+09,
       2.96744727e+10, 1.40022767e+09])
```

```
#Second random forest regressor
rfr_drop = RandomForestRegressor(random_state=42)
rfr_drop.fit(X_train_drop,y_train)
rfr_drop_cv_scores = cross_val_score(rfr_drop, X_train_drop, y_train, scoring='neg_mean_squared_error', cv=10)
rfr_drop_mse_scores = -rfr_drop_cv_scores
rfr_drop_mse_scores
```

```
array([9.89141646e+08, 8.15899050e+08, 8.06379018e+09, 6.23110866e+10,
       1.49205473e+09, 9.57644340e+08, 1.14373324e+10, 2.61438110e+09,
       2.83141180e+10, 1.60552968e+09])
```

In looking at these mean squared error values, notice that, though some of these values are lower than the values received for the linear regression models, the smallest value is over 800,000,000, which is still very large.  Ultimately, these random forest regressors are not very accurate either, and it appears that we will need a more complex model.

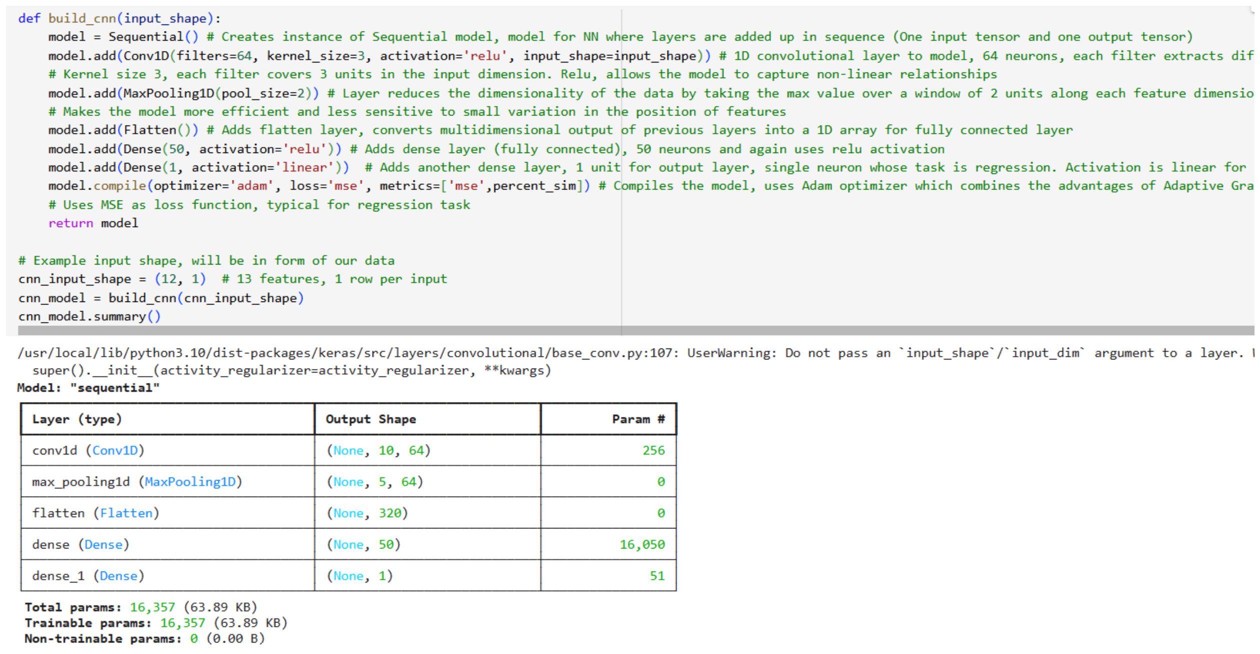
Further, we need to find a way to implement the time series element, so that we can create a model that can predict future housing costs.  For both Linear and Random Forest Regression, there are serious limitations to what we are trying to achieve. For Linear Regression we must assume there is a linear relationship between the input variables and output variables. When it comes to time series data this is never the case because time series data is influenced by factors like trends, seasonality and cycles. Secondly, the model assumes the residuals are independent. In time series data there is often autocorrelation in residuals because past values can be highly predictive of future values. Thirdly, to make linear regression work with time series data you need significant feature engineering. Even with complex features such as lag variables it is not certain that you will capture dependencies or season patterns effectively. Now for Random Forest Regression we run into similar problems. Like linear regression, random forests typically treat each input independently. This means that it does not account for the order of observation which is the crux of time series analysis. Secondly, even if you are able to engineer features to include time steps the feature space can become very high dimensional which will cause a random forest to overfit. Thirdly, random forests and decision trees are not well suited for extrapolation to data ranges beyond the training dataset. Since our task is to forecast future values, random forests may struggle to predict beyond the seen data trends.

For this problem, we thought that using two machine learning models to create a hybrid model would allow us to have the best outcome. Predicting the price of a house is easy, but predicting the future is not so easy.  A Long Short Term Memory model would allow us to capture the temporal features while still leveraging the predictive power of a Convolutional Neural Network on our spatial data. Convolutional Neural Networks are well known for their ability in image processing and computer vision tasks, in our case we want to leverage its potential for handling structured spatial data by leveraging local feature learning. Here we can have it learn how the proximity to amenities correlates with housing prices. It also can integrate those local features into a higher level representation in deeper layers, i.e. understanding complex dependencies such as how the combination of homes age, size and distance to convenience influences the price. Even though the data isn't image based, any multidimensional data that involves spatial relationships can benefit from a CNN's structure. CNN's also have the ability to perform automatic feature detection and extraction which is beneficial to us in this real estate data where determining which features are most predictive of housing price isn't obvious. For these reasons we decided to use a Convolutional Neural Network and a Long Short Term Memory Neural Network in tandem to capture patterns from both of our data sets to create a prediction on what the home value index will be for a house in a specific region.

# Convolutional Neural Network

## Implementation

First, we needed to create a CNN for the spatial data.  A CNN will start with an input layer, then have at least one convolutional layer, followed by at least one fully connected layer and then an output layer.  A convolutional layer will take input in the form of a matrix or "tensor".  It will then convolve and transform the data into a single dimension and then output these results into the next layer.  Here is our original code to make our initial CNN, along with its summary:



```
def build_cnn(input_shape):
    model = Sequential() # Creates instance of Sequential model, model for NN where layers are added up in sequence (One input tensor and one output tensor)
    model.add(Conv1D(filters=64, kernel_size=3, activation='relu', input_shape=input_shape)) # 1D convolutional layer to model, 64 neurons, each filter extracts dif
    # Kernel size 3, each filter covers 3 units in the input dimension. Relu, allows the model to capture non-linear relationships
    model.add(MaxPooling1D(pool_size=2)) # Layer reduces the dimensionality of the data by taking the max value over a window of 2 units along each feature dimensio
    # Makes the model more efficient and less sensitive to small variation in the position of features
    model.add(Flatten()) # Adds flatten layer, converts multidimensional output of previous layers into a 1D array for fully connected layer
    model.add(Dense(50, activation='relu')) # Adds dense layer (fully connected), 50 neurons and again uses relu activation
    model.add(Dense(1, activation='linear'))  # Adds another dense layer, 1 unit for output layer, single neuron whose task is regression. Activation is linear for
    model.compile(optimizer='adam', loss='mse', metrics=['mse',percent_sim]) # Compiles the model, uses Adam optimizer which combines the advantages of Adaptive Gra
    # Uses MSE as loss function, typical for regression task
    return model

# Example input shape, will be in form of our data
cnn_input_shape = (12, 1)  # 13 features, 1 row per input
cnn_model = build_cnn(cnn_input_shape)
cnn_model.summary()
```

```
/usr/local/lib/python3.10/dist-packages/keras/src/layers/convolutional/base_conv.py:107: UserWarning: Do not pass an `input_shape`/`input_dim` argument to a layer. 
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
Model: "sequential"
┌─────────────────────────────────┬────────────────────────┬───────────────┐
│ Layer (type)                    │ Output Shape           │       Param # │
├─────────────────────────────────┼────────────────────────┼───────────────┤
│ conv1d (Conv1D)                 │ (None, 10, 64)         │           256 │
├─────────────────────────────────┼────────────────────────┼───────────────┤
│ max_pooling1d (MaxPooling1D)    │ (None, 5, 64)          │             0 │
├─────────────────────────────────┼────────────────────────┼───────────────┤
│ flatten (Flatten)               │ (None, 320)            │             0 │
├─────────────────────────────────┼────────────────────────┼───────────────┤
│ dense (Dense)                   │ (None, 50)             │        16,050 │
├─────────────────────────────────┼────────────────────────┼───────────────┤
│ dense_1 (Dense)                 │ (None, 1)              │            51 │
└─────────────────────────────────┴────────────────────────┴───────────────┘
 Total params: 16,357 (63.89 KB)
 Trainable params: 16,357 (63.89 KB)
 Non-trainable params: 0 (0.00 B)
```

Notice that this CNN has a 1-dimensional convolutional layer with kernel size 3, some additional layers to pool and flatten the model, a fully connected layer with 50 neurons, and a final output layer with one neuron.  We are using the Adam optimizer, and calculating the mean squared error value, as well as an additional percent_sim value that gets created with the following function:

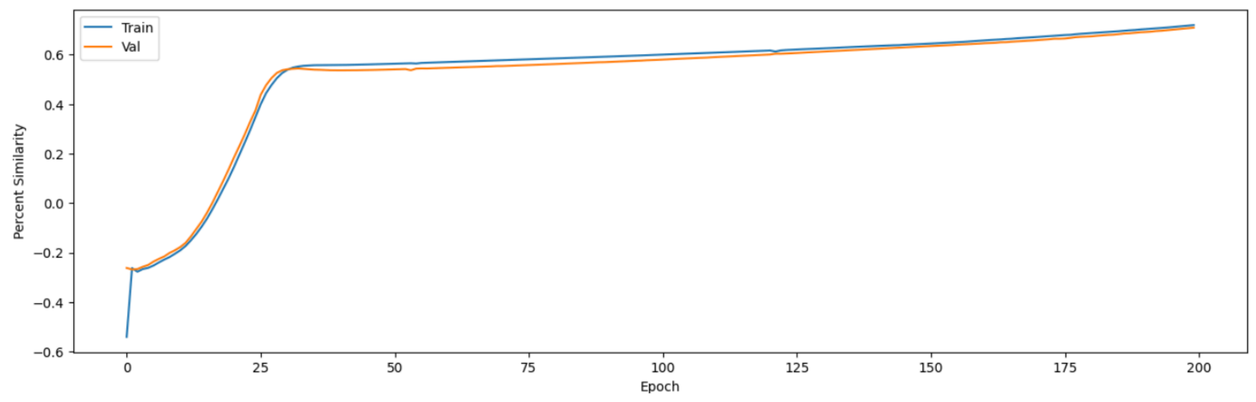
```
keras.utils.get_custom_objects().clear()
@keras.utils.register_keras_serializable(package="my_package", name="percent_sim")
def percent_sim(y_true, y_pred):
  mean = (y_true + y_pred)/2
  diff = abs(y_true - y_pred)
  percent_diff = diff/mean
  return 1 - percent_diff
```

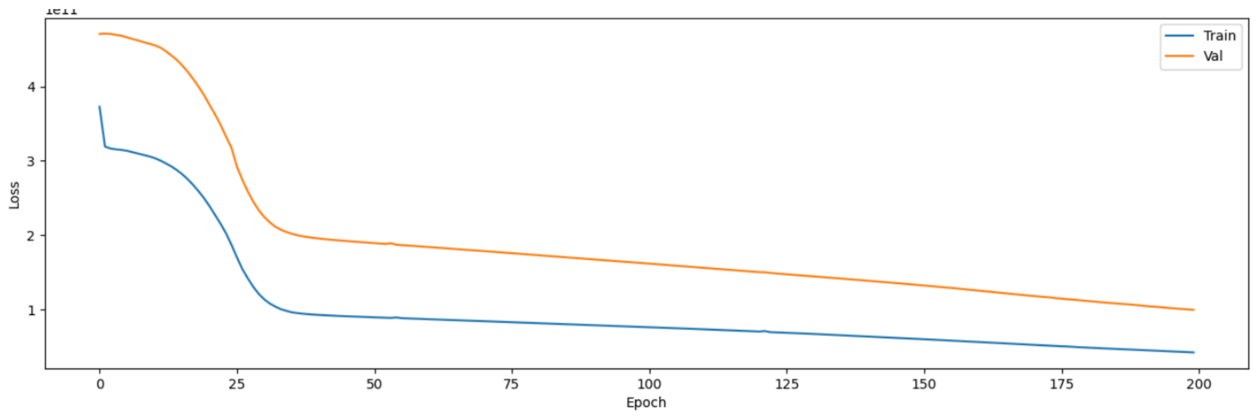
For each epoch, the function will compute the percent difference of the predicted y value and the actual y value by taking the difference of the two values and the average of the two values, then dividing the difference by the average. Since we are creating a metric to determine how accurate the prediction is, we will subtract 1 by this percent difference value to give a measure of "percent similarity", i.e., percent_sim.

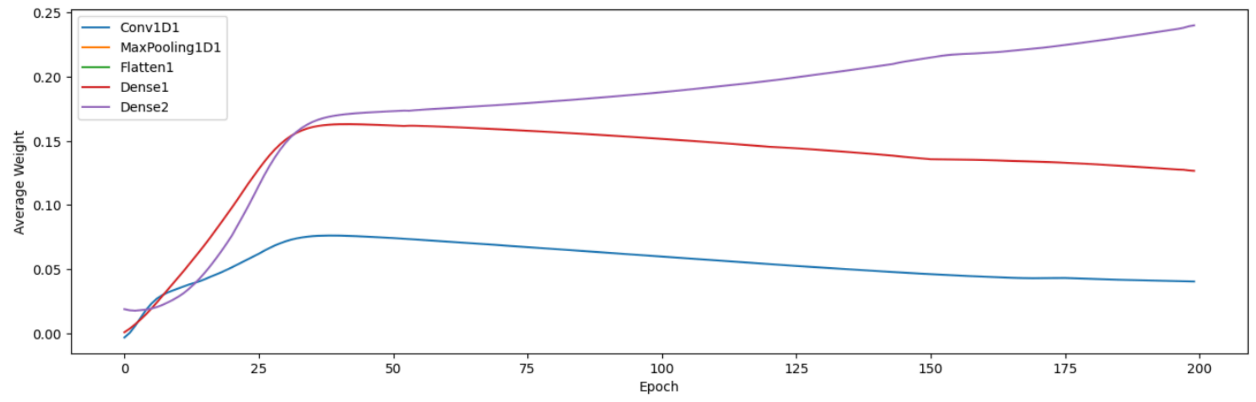
Now, it's time to run this model. We are referencing a callback, customCallback, during the run. This will store the weights of each layer at each epoch, to be viewed and analyzed once the model is run.

```
class customCallback(callbacks.Callback):
  def on_train_begin(self,logs=None):
    self.model.all_weights = []
  def on_epoch_end(self,epoch,logs=None):
    self.weight_list = []
    for layer in self.model.layers:
      if layer.get_weights() == []:
        self.weight_list += [np.nan]
      else:
        self.weight_list += [np.mean(layer.get_weights()[0])]
    self.model.all_weights += [self.weight_list]
```

Here, we have the resulting graphs of the percent_sim, the loss (mean squared error), and average weight per layer vs the epochs:



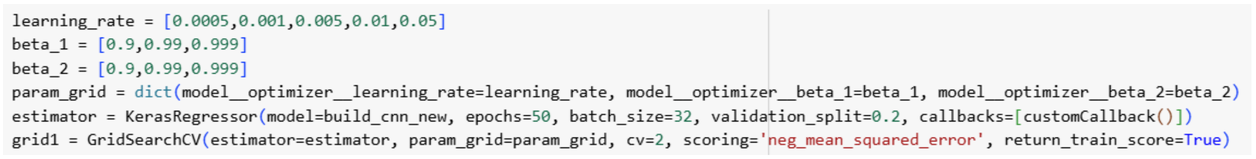




We can see that the percent_sim metric we created increases during the first 30 epochs or so, but then flattens out at around 0.6-0.7. The loss values follow a similar pattern as they decrease at first, but then eventually flatten out. The weights change significantly during those same epochs, but eventually even out. While a 60-70% accuracy is a good start, it is essential to implement improvements to the model to increase this value.

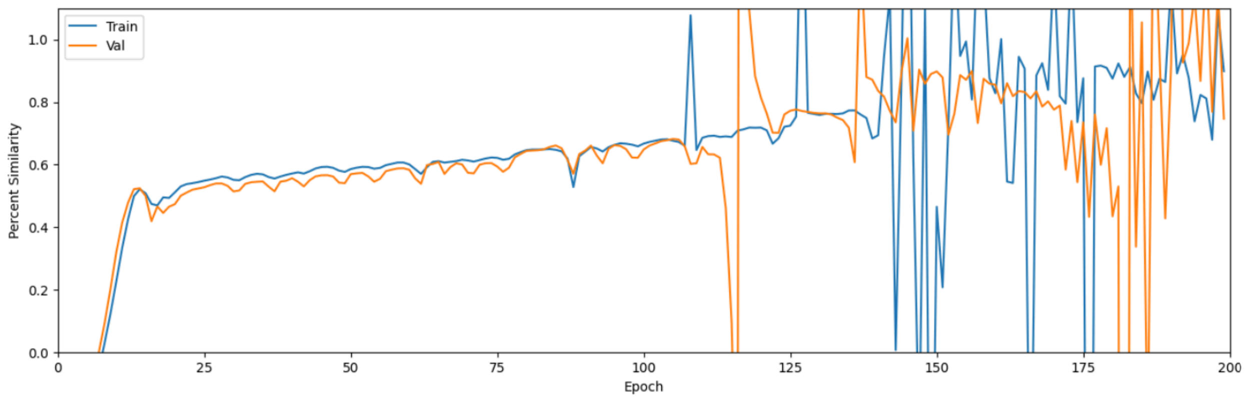
## Refining the Model

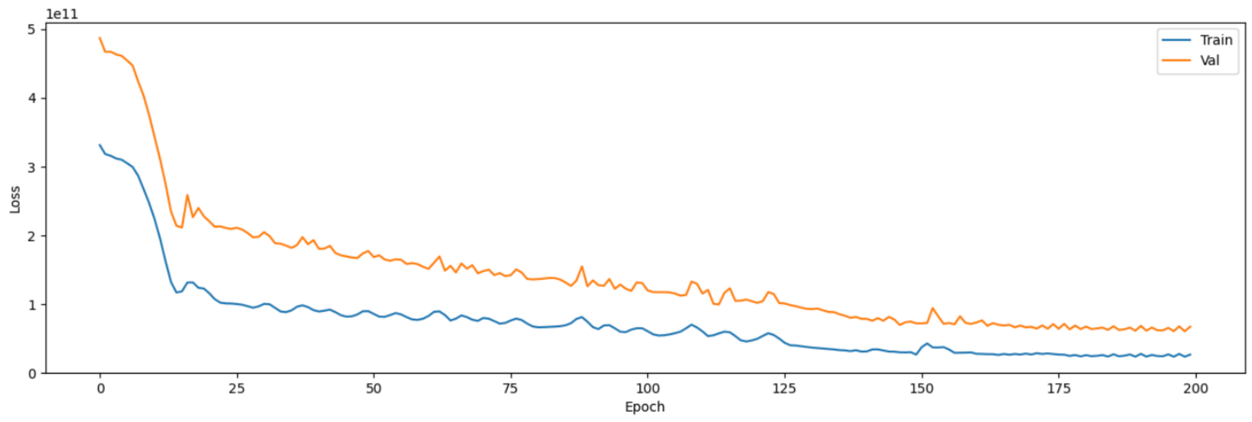
To improve the accuracy of this model, we will start by finding the optimal hyperparameters for our adam optimizer. This includes the learning rate, beta_1 and beta_2.  We need to find the combination of these parameters that will converge at the best rate - preferably pretty quickly.  To do this, we must slightly change the model by setting these hyperparameters to variables that need to be set, and then create a parameter grid with multiple different values for these variables.  This model will then be run multiple times - once for each combination of values.

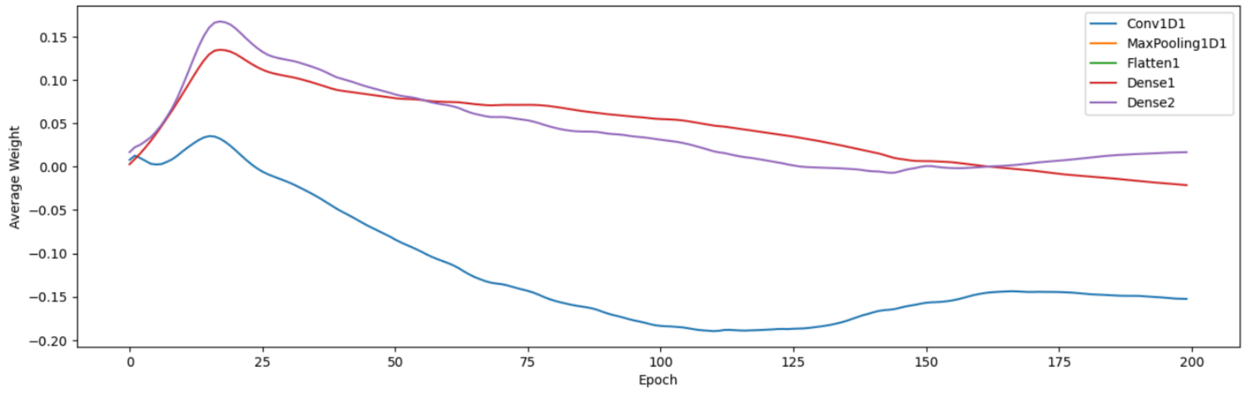


```
learning_rate = [0.0005,0.001,0.005,0.01,0.05]
beta_1 = [0.9,0.99,0.999]
beta_2 = [0.9,0.99,0.999]
param_grid = dict(model__optimizer__learning_rate=learning_rate, model__optimizer__beta_1=beta_1, model__optimizer__beta_2=beta_2)
estimator = KerasRegressor(model=build_cnn_new, epochs=50, batch_size=32, validation_split=0.2, callbacks=[customCallback()])
grid1 = GridSearchCV(estimator=estimator, param_grid=param_grid, cv=2, scoring='neg_mean_squared_error', return_train_score=True)
```

Once run, we will check the models and parameters.  We found that using a learning rate of 0.0005, beta_1 of 0.99, and beta_2 of 0.9 yields the best results with a mean-squared error of 81954160640.  Here are the graphs for this model:







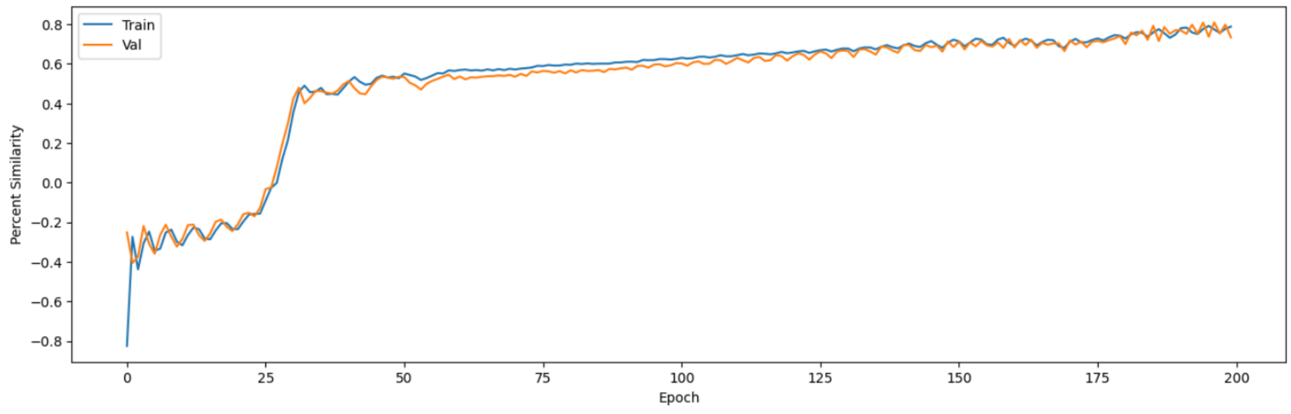
Here we see that the model converges a bit faster, but we are still flattening at around 60-70% accuracy. Eventually the percent_sim values start to vary greatly, and we may need to apply early stopping eventually. But first, it is necessary to apply different tactics to improve the accuracy of the model. Given that the accuracy is stuck, it is likely that the model is underfit. A good strategy for fixing this issue is to add layers. Here is a function that will add a given number of dense layers to the model with a given number of neurons per layer:

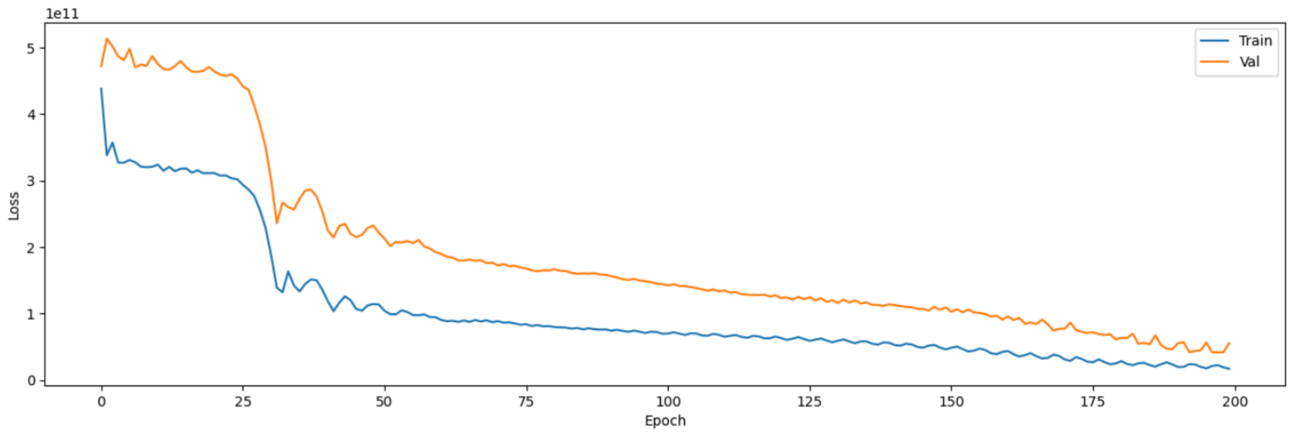
```
@keras.utils.register_keras_serializable(package="my_package", name="add_layer")
def add_layer(model,num_layer,neurons):
  for i in range(num_layer):
    model.add(Dense(neurons,activation='relu'))
```

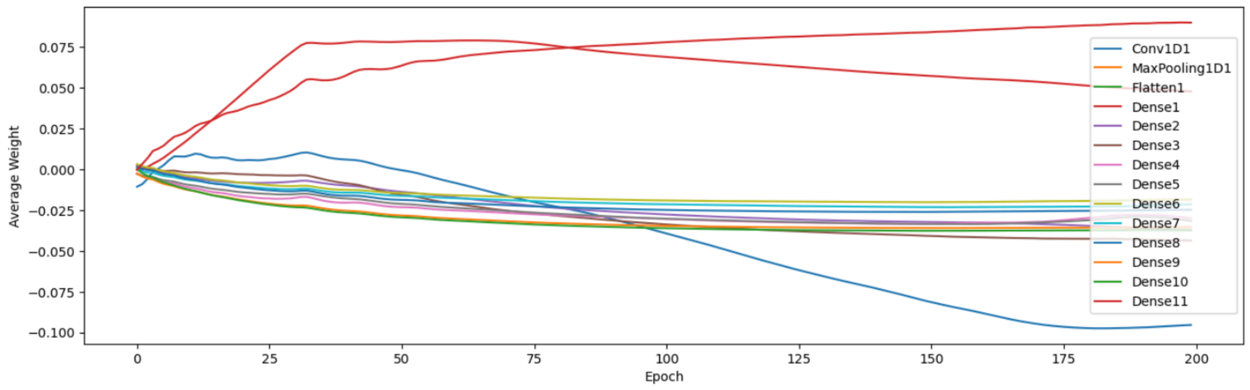
We will now create an updated model utilizing this function to add layers, and we will implement a parameter grid again with multiple values for the 'number of layers' and 'number of neurons' variables, and then run the model with these parameters.

```
num_layers = [1,2,5,10,20]
neurons = [100,50,20]
param_grid2 = dict(model__add_layer__num_layer=num_layers, model__add_layer__neurons=neurons)
estimator2 = KerasRegressor(model=build_cnn_add_layers, epochs=100, batch_size=32, validation_split=0.2, callbacks=[customCallback()])
grid2 = GridSearchCV(estimator=estimator2, param_grid=param_grid2, cv=2, scoring='neg_mean_squared_error', return_train_score=True)
```

We find that the best parameters for the model are 10 layers with 50 neurons added.  See the graphs for this model:







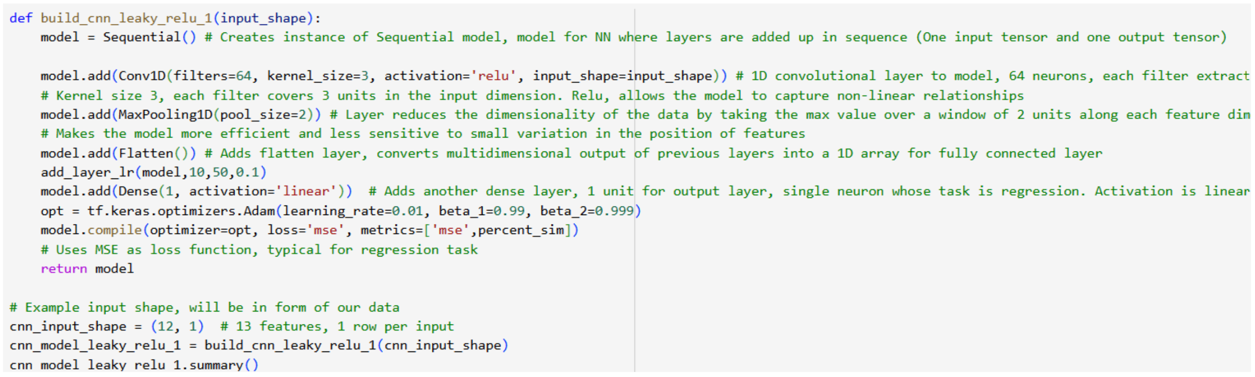
Notice that the accuracy has definitely improved and has reached nearly 80%. The accuracy and loss graphs are pretty smooth. Another thing to note is that the weights seem to flatten out and remain fairly constant as the epochs increase. This could imply that there is a vanishing gradient. To rectify this problem, we will try implementing a leaky ReLU activation function for the hidden dense layers, rather than ReLU. Here is a function to add layers with leaky ReLU activation, with number of layers, number of neurons, and the slope as parameters.

```
@keras.utils.register_keras_serializable(package="my_package", name="add_layer_lr")
def add_layer_lr(model,num_layer,neurons,slope):
  act = tf.keras.layers.LeakyReLU(negative_slope=slope)
  for i in range(num_layer):
    model.add(Dense(neurons,activation=act))
```

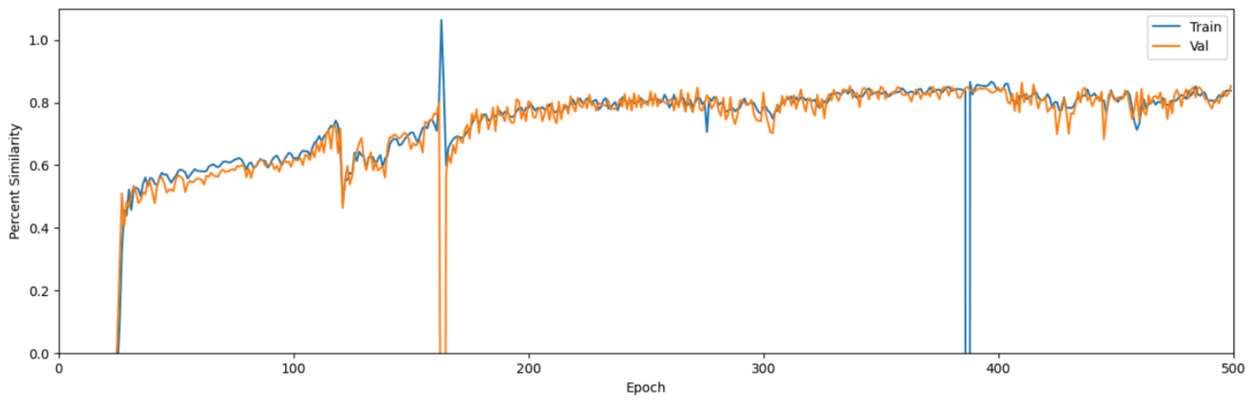
We will now update our model to use this function instead of our original add_layer function, then create a parameter grid with various slope values, and then run the model with these parameters.

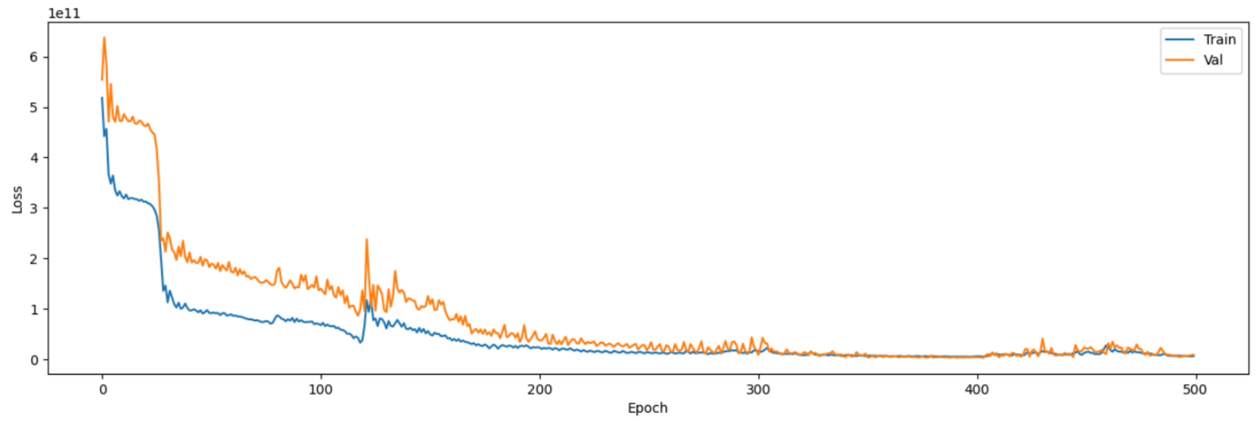
```
slope = [0.1,0.3,0.5,0.8]
param_grid3 = dict(model__add_layer__slope=slope)
estimator3 = KerasRegressor(model=build_cnn_leaky_relu, epochs=100, batch_size=32, validation_split=0.2, callbacks=[customCallback()])
grid3 = GridSearchCV(estimator=estimator3, param_grid=param_grid3, cv=3, scoring='neg_mean_squared_error', return_train_score=True)
```

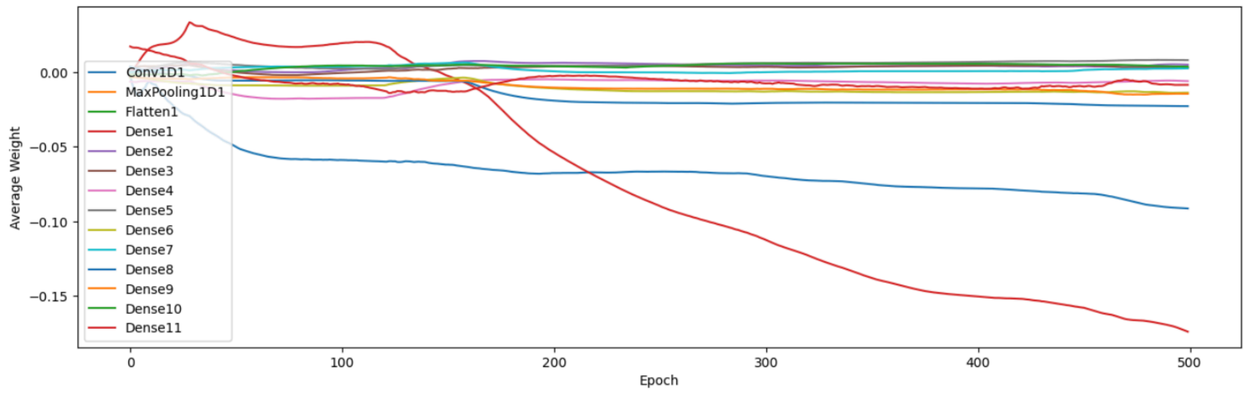
We found that the best parameters for this model have a slope value of 0.1.  Here is the final model, the graphs, and a sample of the percent_sim and loss values for the last epochs:

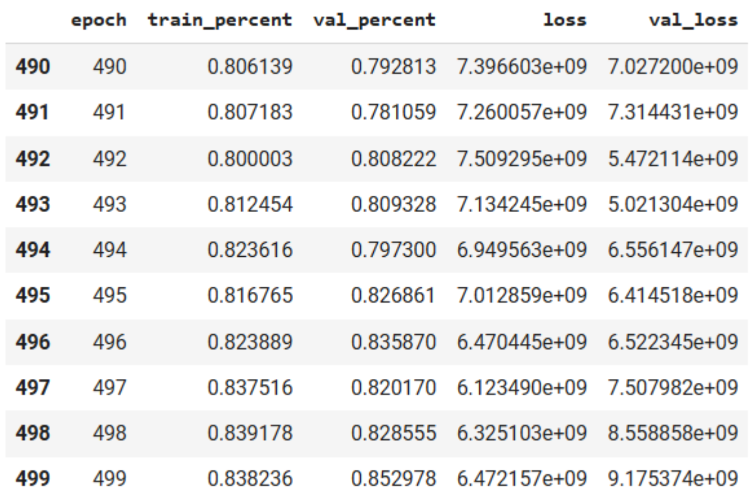


```
def build_cnn_leaky_relu_1(input_shape):
    model = Sequential() # Creates instance of Sequential model, model for NN where layers are added up in sequence (One input tensor and one output tensor)

    model.add(Conv1D(filters=64, kernel_size=3, activation='relu', input_shape=input_shape)) # 1D convolutional layer to model, 64 neurons, each filter extract
    # Kernel size 3, each filter covers 3 units in the input dimension. Relu, allows the model to capture non-linear relationships
    model.add(MaxPooling1D(pool_size=2)) # Layer reduces the dimensionality of the data by taking the max value over a window of 2 units along each feature dim
    # Makes the model more efficient and less sensitive to small variation in the position of features
    model.add(Flatten()) # Adds flatten layer, converts multidimensional output of previous layers into a 1D array for fully connected layer
    add_layer_lr(model,10,50,0.1)
    model.add(Dense(1, activation='linear'))  # Adds another dense layer, 1 unit for output layer, single neuron whose task is regression. Activation is linear
    opt = tf.keras.optimizers.Adam(learning_rate=0.01, beta_1=0.99, beta_2=0.999)
    model.compile(optimizer=opt, loss='mse', metrics=['mse',percent_sim])
    # Uses MSE as loss function, typical for regression task
    return model

# Example input shape, will be in form of our data
cnn_input_shape = (12, 1)  # 13 features, 1 row per input
cnn_model_leaky_relu_1 = build_cnn_leaky_relu_1(cnn_input_shape)
cnn_model_leaky_relu_1.summary()
```









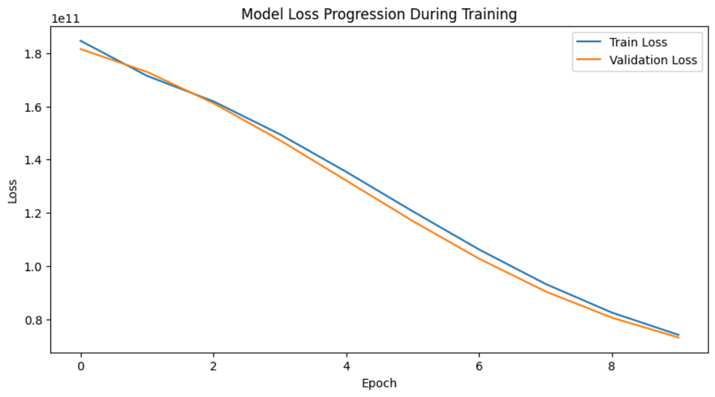
| | epoch | train_percent | val_percent | loss | val_loss |
|---|---|---|---|---|---|
| **490** | 490 | 0.806139 | 0.792813 | 7.396603e+09 | 7.027200e+09 |
| **491** | 491 | 0.807183 | 0.781059 | 7.260057e+09 | 7.314431e+09 |
| **492** | 492 | 0.800003 | 0.808222 | 7.509295e+09 | 5.472114e+09 |
| **493** | 493 | 0.812454 | 0.809328 | 7.134245e+09 | 5.021304e+09 |
| **494** | 494 | 0.823616 | 0.797300 | 6.949563e+09 | 6.556147e+09 |
| **495** | 495 | 0.816765 | 0.826861 | 7.012859e+09 | 6.414518e+09 |
| **496** | 496 | 0.823889 | 0.835870 | 6.470445e+09 | 6.522345e+09 |
| **497** | 497 | 0.837516 | 0.820170 | 6.123490e+09 | 7.507982e+09 |
| **498** | 498 | 0.839178 | 0.828555 | 6.325103e+09 | 8.558858e+09 |
| **499** | 499 | 0.838236 | 0.852978 | 6.472157e+09 | 9.175374e+09 |

Now, we see the accuracy is over 80% after 500 epochs, which is the highest it has been so far. We can now use this final CNN model for handling our spatial data.

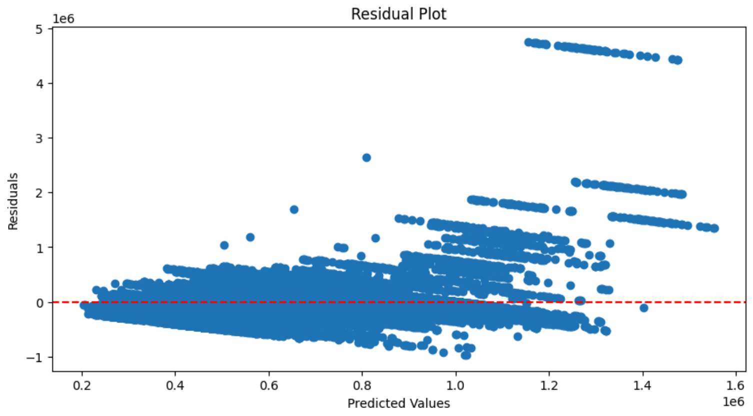
## Long Short Term Memory Neural Network

### Implementation

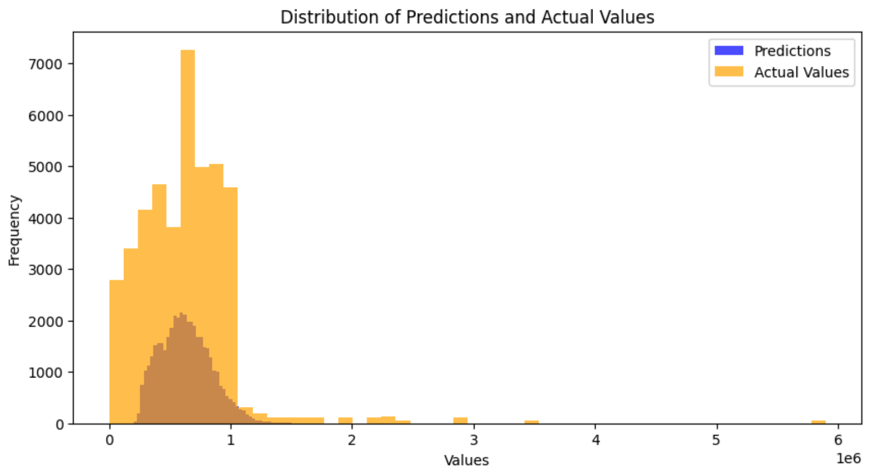
After processing the temporal data as well as feature selection it was time to create the LSTM model. Initially we began with a basic LSTM layer with 50 neurons to capture the temporal dependencies in the data. The initial model configuration targeted a single output layer with a linear activation function whose job it was to predict continuous values of house prices. The mean squared error was used as the loss function as well as the Adam optimizer for its efficiency in handling sparse gradients and adaptive learning rate capabilities. A hold out validation set was used to monitor the models performance and prevent overfitting. This was done by using train test split to ensure the model could generalize well on new unseen data. The input was a time sequence of 12 steps, or a year and the target was the most recent 6 months of data.



From this loss graph you can see that the loss is continuously decreasing for both the training and validation data, this is a great sign for us because it means we can continue to decrease the loss by adding more epochs and by improving model complexity.



From this residual plot of our basic model we can draw a few conclusions. There are a good amount of residuals clustered near zero but there are also a good amount that are not, indicating bad predictions. The patterns suggest a problem with heteroscedasticity, so the variance of the residuals is not consistent across all levels of predicted values. There are also several outliers, especially for higher predicted values. These indicate instances where the model predictions deviate significantly from the actual values. The clustering of residuals around specific bands suggest that the model may be systematically over or underestimating across certain ranges of the data pointing us to potential model bias in high value predictions.



This graph shows us the distribution of the predicted values vs the actual values. Here the predicted values are behind the actual values so they look dark orange. As you can see the model got the overall shape somewhat right, but we still have a long way before we can say our models' predictions are accurate.

**Refining the Model**

From the graphs we just looked at we took the insight and attempted to refine the model in a way that would address heteroscedasticity and improve the model complexity so our predictions would be more similar to the actual values. To handle the underfitting observations from the initial predictions we increased the model complexity by adding more LSTM neurons and by adding a dense layer with 256 neurons to capture more complex relationships in the sequence data. We then added dropout layers to reduce overfitting by randomly dropping neurons in the neural network during training. We also implemented a keras turner to optimize the models architecture and learning parameters. We found that the best values were 100 units, a dropout rate of .2, and a 256 unit dense layer which solidified our ideas behind reducing the overfitting and underfitting of the first model. We also created a new custom metric for evaluating how well the model was doing called average difference metric which took the actual value of the house and subtracted the predicted then averaged it among all the values! This was extremely helpful during training because we could see in real time if our model was learning well!

## Hybrid Model

Now that the CNN and LSTM models have both been trained and refined, it's time to combine them into a hybrid model. That way, we will finally be able to forecast the price of a house in the future based on its various features.

To do this, we tried to combine our predictions we got from our separate CNN model and LSTM model. This proved to be very difficult, and we were unable to get any meaningful results with this approach. So instead, we merged our spatial and temporal datasets into one large dataset by the city/region. Our final data had, for each city, the full spatial set for that city for each time and HVI record for the city.

The hybrid model concatenates the layers from the refined CNN and LSTM models. It then has another hidden dense layer with 256 neurons and ReLU activation, and a final output layer with 1 neuron and linear activation.

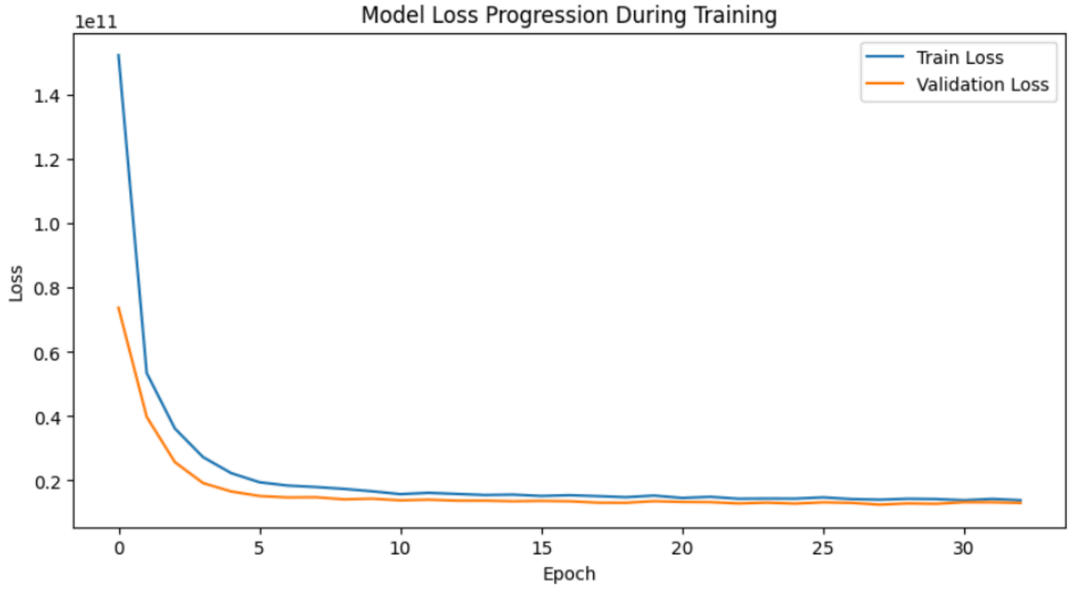
```python
# Define the model architecture with dropout and batch normalization
def build_model():
    cnn_input = Input(shape=(X_train.shape[1], 1))
    x = Conv1D(filters=64, kernel_size=3, activation='relu')(cnn_input)
    x = MaxPooling1D(pool_size=2)(x)
    x = BatchNormalization()(x)
    x = Dropout(0.3)(x)
    x = Flatten()(x)
    add_layer_lr(x,10,50,0.1)

    lstm_input = Input(shape=(sequence_length, 1))
    y = LSTM(100, activation='relu', dropout=0.2)(lstm_input)

    # Combine CNN and LSTM outputs
    combined = concatenate([x, y])

    combined = Dense(256, activation='relu')(combined)  # Using 256 dense units

    output = Dense(1, activation='linear')(combined)
    model = Model(inputs=[cnn_input, lstm_input], outputs=output)
    opt = tf.keras.optimizers.Adam(learning_rate=0.005, beta_1=0.9, beta_2=0.9)
    model.compile(optimizer=opt, loss='mean_squared_error', metrics=['mean_absolute_error', avg_diff_metric])
    return model
```

Notice, we are using a avg_diff_metric function to evaluate our accuracy, which is as follows:

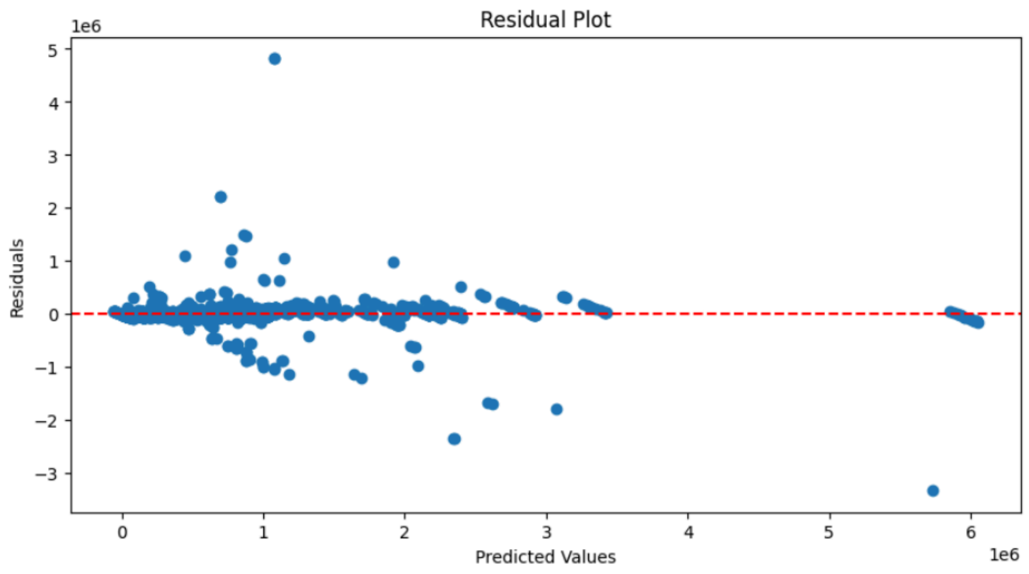
```python
# Define custom metric
def avg_diff_metric(y_true, y_pred):
    return K.mean(K.abs(y_pred - y_true))
```

We can now run our combined spatial and temporal dataset through this model.

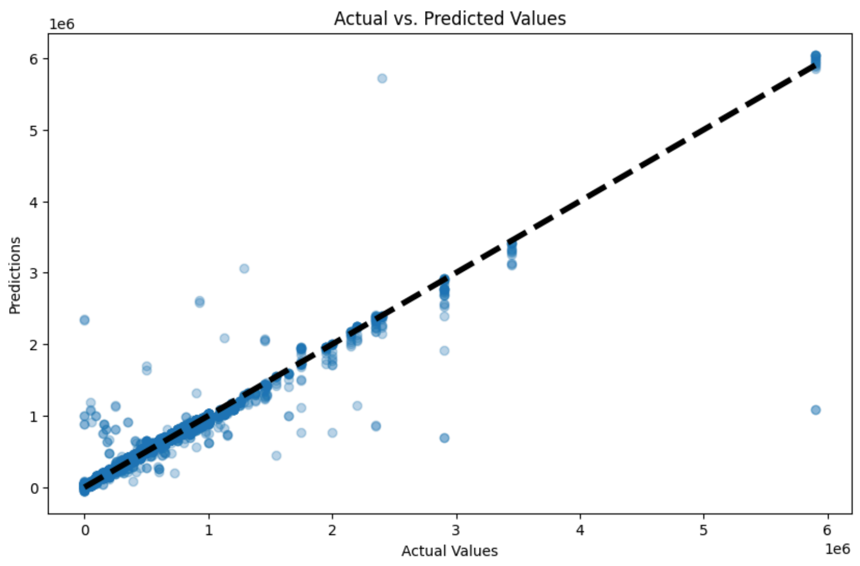
```
Test MSE: 12508803072.0
Average Difference: 30404.162109375
```



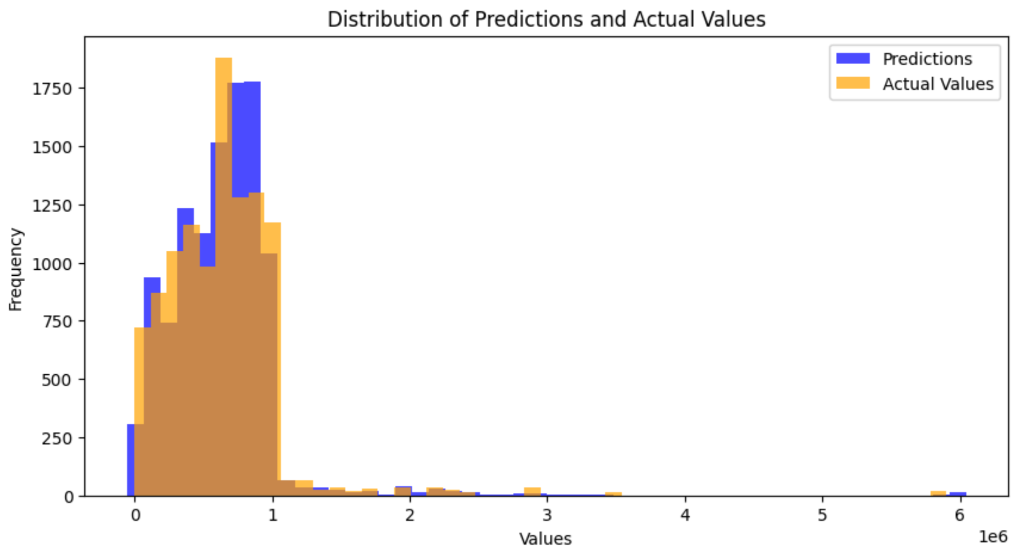
We see that the loss function decreases rapidly.  When evaluating the model on the test set, we get a mean square error of 12508803072, which is the best we have seen, and an average difference of 30404, which is pretty close when it comes to predicting housing prices. Here is a plot of the residuals:



This looks much much better than the first residual plot we looked at. You can see that there are outliers but nowhere near as many as before. The spread as the value increases tells us the model is less accurate with higher predictions.



This graph shows a plot of the actual vs predicted values where being closer to the line means the prediction was more accurate. There are a couple of outliers mainly at the mid price range but the lower end of the spectrum was very accurate.



Here is the distribution plot of the final model and you can see that it is much more accurate than the first distribution plot. Seeing that the predicted values are narrower than the actual around the mode tells us that the model is being somewhat conservative. The peak of the predicted is a bit wider and to the right than the actual peak which tells us that the model still may be overfitting to a common value.

## Conclusion

In this project we set out to develop a hybrid model that leveraged both spatial and temporal features of housing price data. This model, consisting of a Convolutional Neural Network (CNN) and Long Short-Term Memory (LSTM) Neural Network, was designed to harness the strength of CNN's feature detection and LSTM's sequence prediction capabilities. Throughout the development process, we fine-tuned the individual models, which then ultimately combined the two to achieve a very impressive average difference of $30,404 USD.

However, this is not the endpoint of our journey. To achieve better results, we believe improvements can be made by enhancing the quality and variety of data used. During discussions with my uncle, who owns a real estate group in Arizona, it was suggested that we could add more spatial features such as distance to golf courses, change from city designations to zip codes, include whether there are HOA dues, and add housing types such as single-family homes, condos, and townhouses, multi-family homes, etc.

Arguably, the most significant enhancement would be to include precise housing prices for every house at every time point where we currently have the average home value index of the region. This way the LSTM would have another temporal component to make predictions off of which I believe would increase the prediction accuracy even further.

Ultimately, this model has potential to pivot into a tool that uses the same models but has a binary classification output that can predict which direction the price of the house will go, up or down. Such a tool could be invaluable to real estate groups who are planning on expanding into new regions and require data driven insight into which regions are likely to experience price increases. By continuously refining our model and incorporating more data, we can significantly enhance its usability and accuracy making it a robust decision making tool in the real estate industry.

## Sources

Zillow Group, Inc. (n.d.). *Discover A place you'll love to live*. Trulia Real Estate Search. https://www.trulia.com/

Zillow Group, Inc. (n.d.). Housing Data. https://www.zillow.com/research/data/

SupermarketPage.com. (2001-2024). *Arizona Supermarkets*. Supermarket Page. (2001). http://www.supermarketpage.com/state/AZ/

Howard, J. (2021, June 11). *AZGeo Data Hub*. Arizona Schools. https://azgeo-data-hub-agic.hub.arcgis.com/datasets/29abc422fd0541f2b3bbe21d1b16c5f6/explore

World Population Review. (2024). *Arizona cities by population (2024)*. World Population Review. https://worldpopulationreview.com/states/cities/arizona

www.gps-coordinates.net. (2024). *GPS coordinates converter*. GPS-coordinates. https://www.gps-coordinates.net/gps-coordinates-converter

Kettle, S. (2017, October 5). *Distance on a sphere: The Haversine formula*. Esri Community. https://community.esri.com/t5/coordinate-reference-systems-blog/distance-on-a-sphere-the-haversine-formula/ba-p/902128#:~:text=For%20example%2C%20haversine(%CE%B8),longitude%20of%20the%20two%20points.