```
!pip install umap-learn
```

```
Collecting umap-learn
  Downloading umap-learn-0.5.5.tar.gz (90 kB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 90.9/90.9 kB 1.3 MB/s eta 0:00:00
  Preparing metadata (setup.py) ... done
Requirement already satisfied: numpy>=1.17 in /usr/local/lib/python3.10/dist-packages (from umap-learn) (1.23.5)
Requirement already satisfied: scipy>=1.3.1 in /usr/local/lib/python3.10/dist-packages (from umap-learn) (1.11.3)
Requirement already satisfied: scikit-learn>=0.22 in /usr/local/lib/python3.10/dist-packages (from umap-learn) (1.2.2)
Requirement already satisfied: numba>=0.51.2 in /usr/local/lib/python3.10/dist-packages (from umap-learn) (0.58.1)
Collecting pynndescent>=0.5 (from umap-learn)
  Downloading pynndescent-0.5.11-py3-none-any.whl (55 kB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 55.8/55.8 kB 6.7 MB/s eta 0:00:00
Requirement already satisfied: tqdm in /usr/local/lib/python3.10/dist-packages (from umap-learn) (4.66.1)
Requirement already satisfied: llvmlite<0.42,>=0.41.0dev0 in /usr/local/lib/python3.10/dist-packages (from numba>=0.51.2->umap-learn) (0
Requirement already satisfied: joblib>=0.11 in /usr/local/lib/python3.10/dist-packages (from pynndescent>=0.5->umap-learn) (1.3.2)
Requirement already satisfied: threadpoolctl>=2.0.0 in /usr/local/lib/python3.10/dist-packages (from scikit-learn>=0.22->umap-learn) (3.
Building wheels for collected packages: umap-learn
  Building wheel for umap-learn (setup.py) ... done
  Created wheel for umap-learn: filename=umap_learn-0.5.5-py3-none-any.whl size=86831 sha256=22a2408bbb51dfeb32e14b4d4dc13a66c88b757bbf8
  Stored in directory: /root/.cache/pip/wheels/3a/70/07/428d2b58660a1a3b431db59b806a10da736612ebbc66c1bcc5
Successfully built umap-learn
Installing collected packages: pynndescent, umap-learn
Successfully installed pynndescent-0.5.11 umap-learn-0.5.5
```

```
import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt

from sklearn.decomposition import PCA
from sklearn.preprocessing import StandardScaler
import umap
from sklearn.manifold import TSNE

from sklearn.linear_model import LinearRegression

import sklearn.metrics as metrics

from sklearn.model_selection import train_test_split
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import GridSearchCV

from sklearn.tree import DecisionTreeRegressor
from sklearn.ensemble import RandomForestRegressor
```

```
boston_housing=pd.read_csv("BostonHousing.csv",header=0)
```

```
boston_housing.head()
```

	crim	zn	indus	chas	nox	rm	age	dis	rad	tax	ptratio	b	lstat
0	0.00632	18.0	2.31	0	0.538	6.575	65.2	4.0900	1	296	15.3	396.90	4.98
1	0.02731	0.0	7.07	0	0.469	6.421	78.9	4.9671	2	242	17.8	396.90	9.14
2	0.02729	0.0	7.07	0	0.469	7.185	61.1	4.9671	2	242	17.8	392.83	4.03
3	0.03237	0.0	2.18	0	0.458	6.998	45.8	6.0622	3	222	18.7	394.63	2.94
4	0.06905	0.0	2.18	0	0.458	7.147	54.2	6.0622	3	222	18.7	396.90	5.33

```
boston_housing.dtypes
```

```
crim       float64
zn         float64
indus      float64
chas         int64
nox        float64
rm         float64
age        float64
dis        float64
```

```
rad          int64
tax          int64
ptratio    float64
b          float64
lstat      float64
medv       float64
dtype: object
```

```
boston_housing.isnull().sum()
```

```
crim       0
zn         0
indus      0
chas       0
nox        0
rm         5
age        0
dis        0
rad        0
tax        0
ptratio    0
b          0
lstat      0
medv       0
dtype: int64
```

```
boston_housing['rm'].index[boston_housing['rm'].apply(np.isnan)]
```

```
Int64Index([10, 35, 63, 96, 135], dtype='int64')
```

```
boston_housing.describe()
```

	crim	zn	indus	chas	nox	rm	age
count	506.000000	506.000000	506.000000	506.000000	506.000000	501.000000	506.000000
mean	3.613524	11.363636	11.136779	0.069170	0.554695	6.284341	68.574901
std	8.601545	23.322453	6.860353	0.253994	0.115878	0.705587	28.148861
min	0.006320	0.000000	0.460000	0.000000	0.385000	3.561000	2.900000
25%	0.082045	0.000000	5.190000	0.000000	0.449000	5.884000	45.025000
50%	0.256510	0.000000	9.690000	0.000000	0.538000	6.208000	77.500000
75%	3.677083	12.500000	18.100000	0.000000	0.624000	6.625000	94.075000
max	88.976200	100.000000	27.740000	1.000000	0.871000	8.780000	100.000000

At this point we have so far seen the structure of the overall data. We see that the 'rm' column has 5 null/NaN values which will need to be addressed. The types seem to have been inferred correctly as we see no object type columns meaning all columns contain values of same type

```
boston_housing.median()
```

```
crim         0.25651
zn           0.00000
indus        9.69000
chas         0.00000
nox          0.53800
rm           6.20800
age         77.50000
dis          3.20745
rad          5.00000
tax        330.00000
ptratio     19.05000
b          391.44000
lstat       11.36000
medv        21.20000
dtype: float64
```

The mean and median of 'rm' are very similar so it could be a good try to replace the 5 missing values with either mean or median.

```
boston_housing['rm'].fillna(boston_housing['rm'].median(), inplace=True)
```

```
boston_housing.isnull().sum()
```

```
crim       0
zn         0
indus      0
chas       0
nox        0
rm         0
age        0
dis        0
rad        0
tax        0
ptratio    0
b          0
lstat      0
medv       0
dtype: int64
```

EDA

Analyze any type of relationship within or between features

```
target_variable = boston_housing.iloc[:, -1]
features = boston_housing.iloc[:, :-1]
```

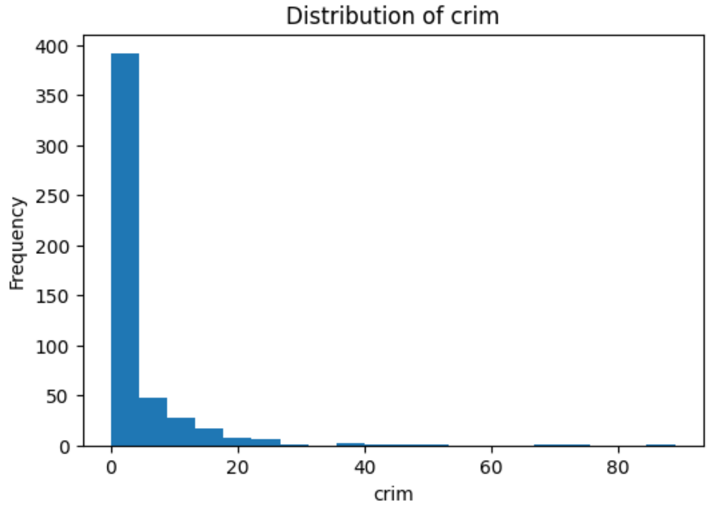
```
target_variable.head()
```

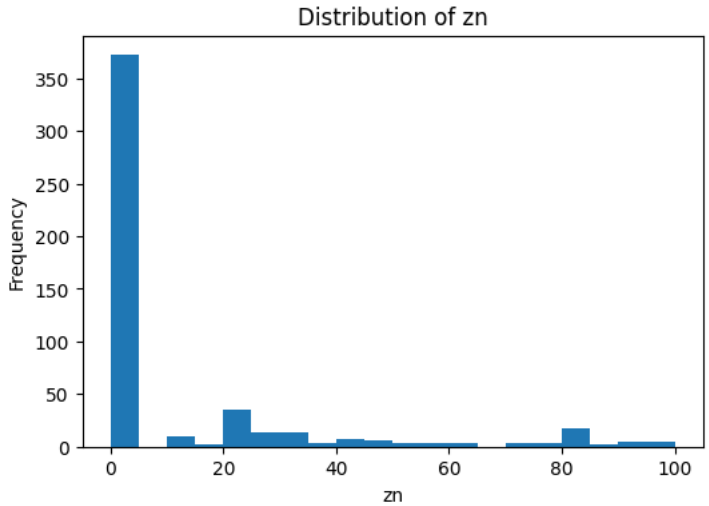
```
0    24.0
1    21.6
2    34.7
3    33.4
4    36.2
Name: medv, dtype: float64
```

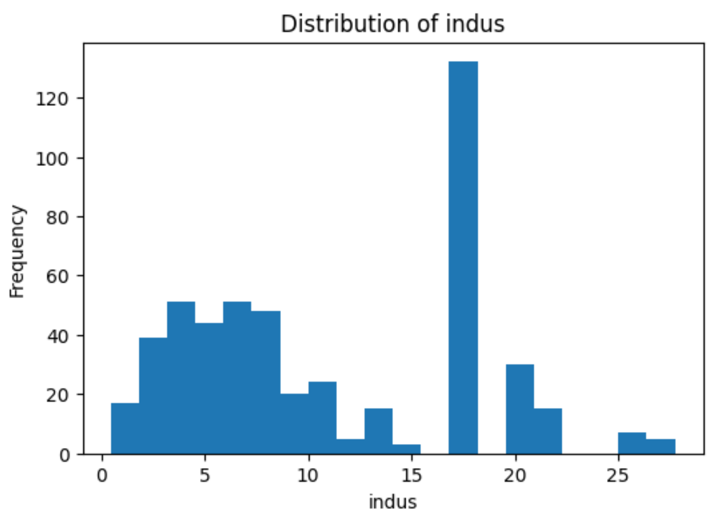
```
features.head()
```

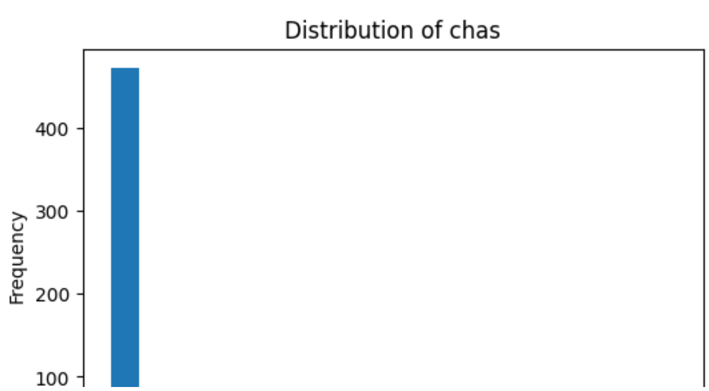
	crim	zn	indus	chas	nox	rm	age	dis	rad	tax	ptratio	b	lstat
0	0.00632	18.0	2.31	0	0.538	6.575	65.2	4.0900	1	296	15.3	396.90	4.98
1	0.02731	0.0	7.07	0	0.469	6.421	78.9	4.9671	2	242	17.8	396.90	9.14
2	0.02729	0.0	7.07	0	0.469	7.185	61.1	4.9671	2	242	17.8	392.83	4.03
3	0.03237	0.0	2.18	0	0.458	6.998	45.8	6.0622	3	222	18.7	394.63	2.94
4	0.06905	0.0	2.18	0	0.458	7.147	54.2	6.0622	3	222	18.7	396.90	5.33

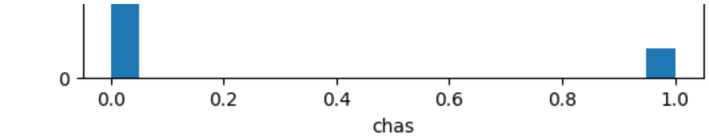
```
# Plotting the distribution of each feature in separate graphs
for column in features.columns:
    plt.figure(figsize=(6, 4))  # Adjust the figure size if needed
    plt.hist(features[column], bins=20)  # You can also use sns.distplot for more detailed plots
    plt.title(f"Distribution of {column}")
    plt.xlabel(column)
    plt.ylabel("Frequency")
    plt.show()
```


Distribution of crim
Frequency
crim
0 50 100 150 200 250 300 350 400
0 20 40 60 80

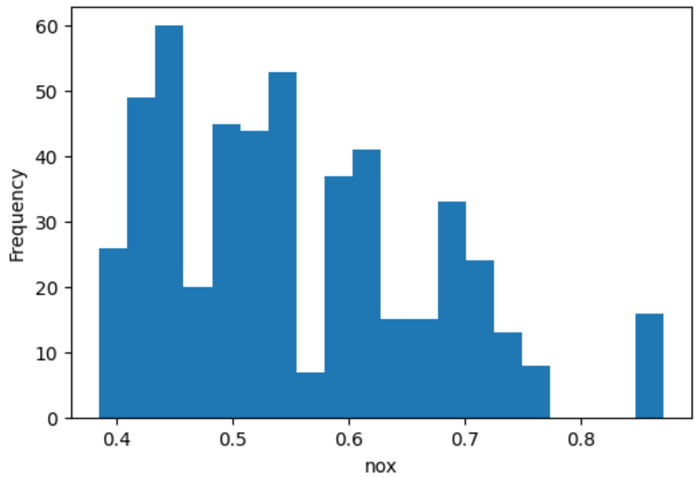

Distribution of zn
Frequency
zn
0 50 100 150 200 250 300 350
0 20 40 60 80 100


Distribution of indus
Frequency
indus
0 20 40 60 80 100 120
0 5 10 15 20 25

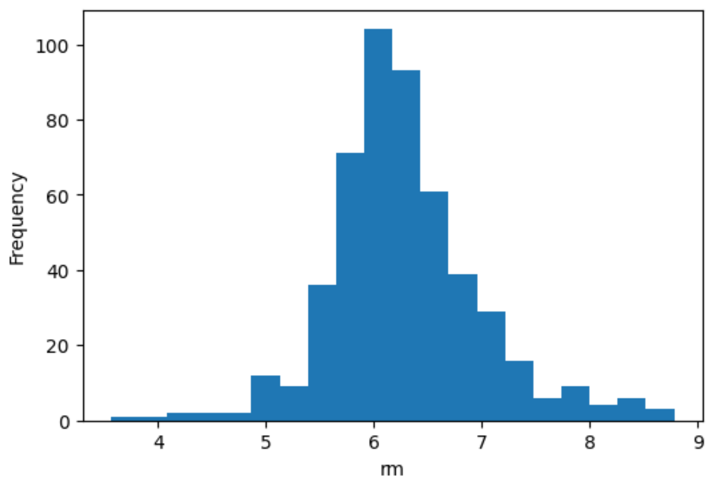

Distribution of chas
Frequency
100 200 300 400



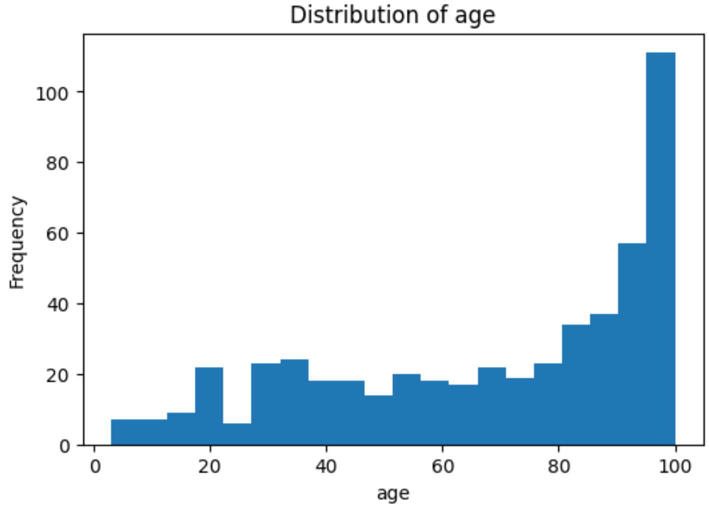
Distribution of nox



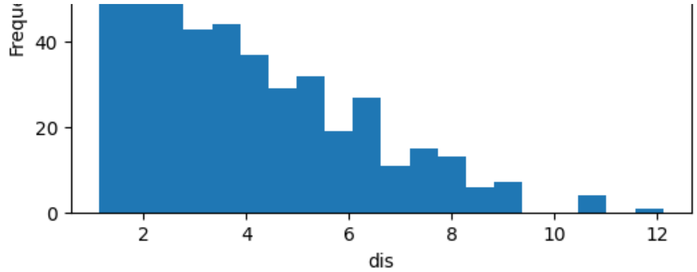
Distribution of rm



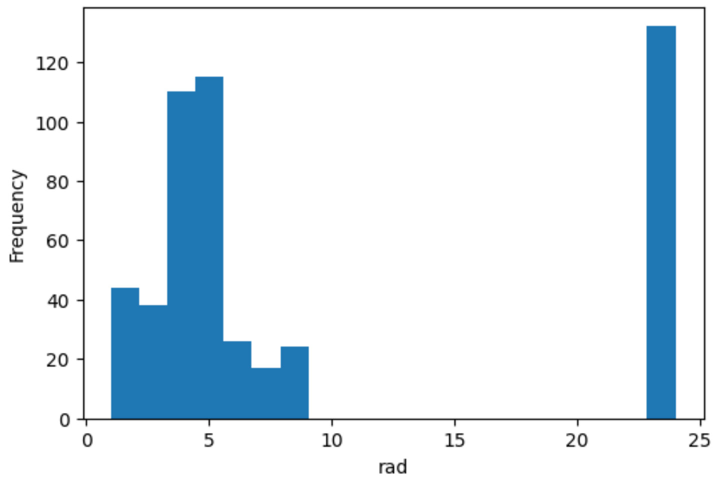
Distribution of age



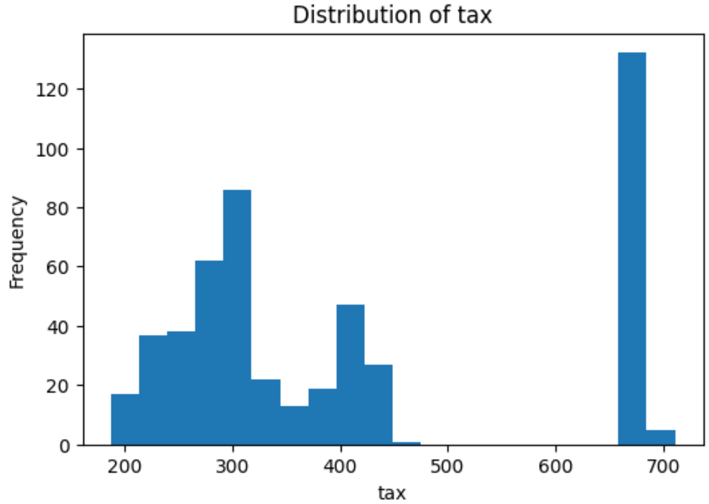
Distribution of dis



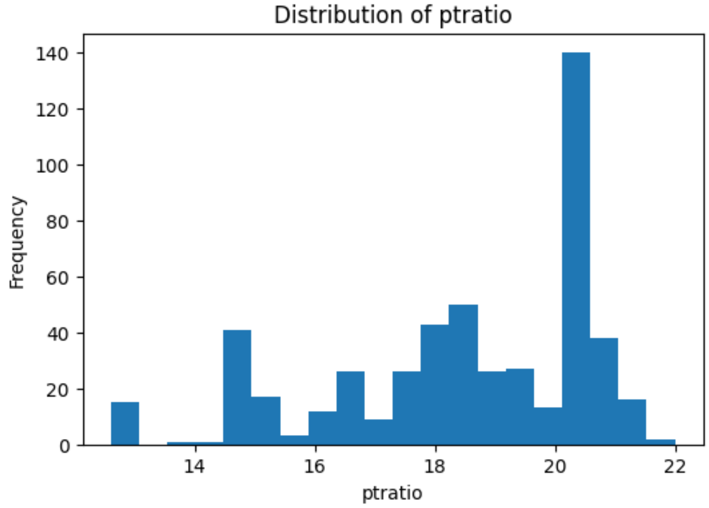
Distribution of rad



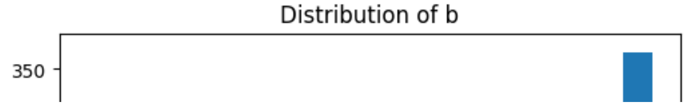
Distribution of tax

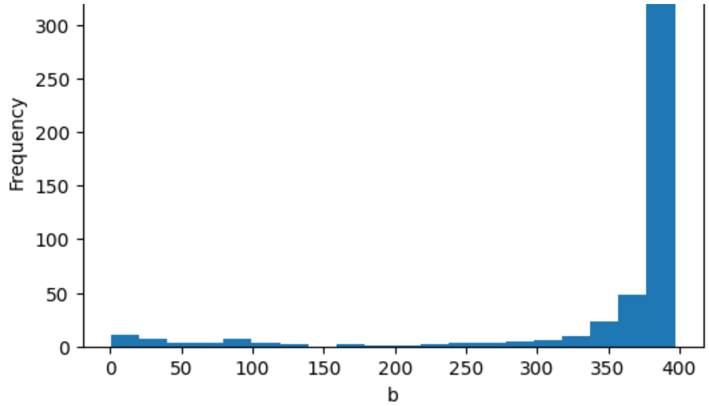


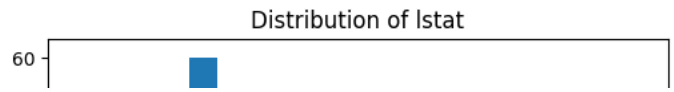
Distribution of ptratio



Distribution of b

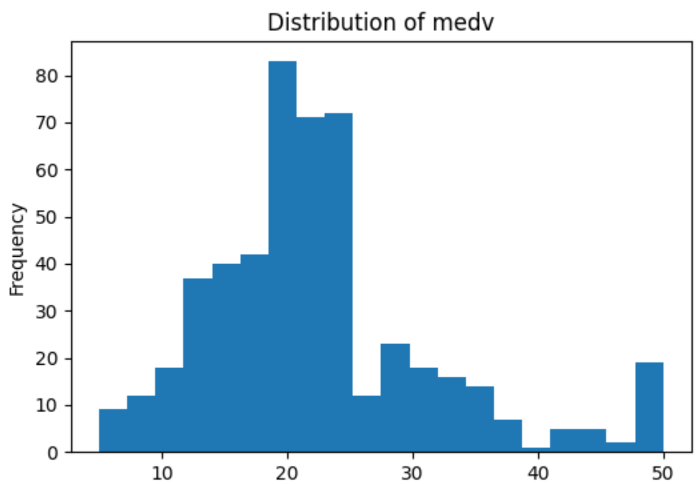






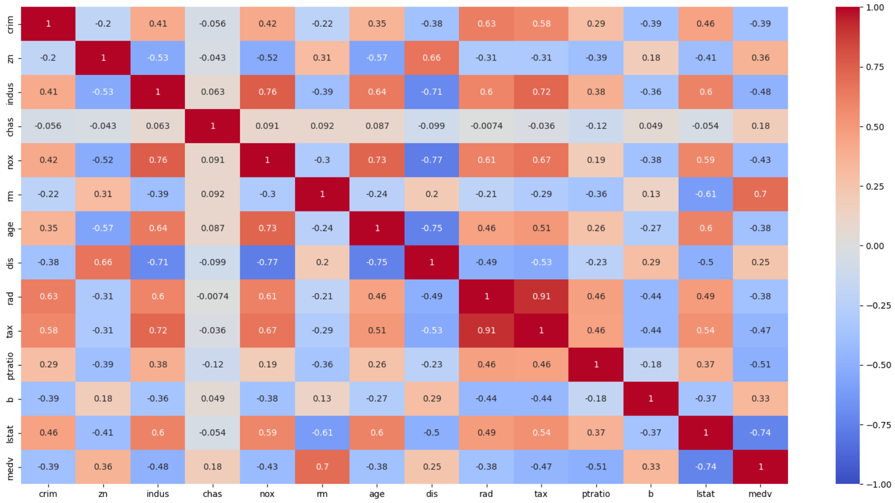
the columns crim, zn, and b have highly skewed data. Chas is apparent now as a 'categorical' column

```
plt.figure(figsize=(6, 4))  # Adjust the figure size if needed
plt.hist(target_variable, bins=20)  # You can also use sns.distplot for more detailed plots
plt.title("Distribution of medv")
plt.ylabel("Frequency")
plt.show()
```



correlation analysis

```
plt.figure(figsize=(20, 10))
sns.heatmap(boston_housing.corr(),annot=True,cmap="coolwarm",center=0,vmin=-1,vmax=1);
```



'chas' doesnt seem to be correlated to anything (could possibly drop) however 'tax' and 'rad' seem to have very high correlation.

Dimensionality Visual Analysis

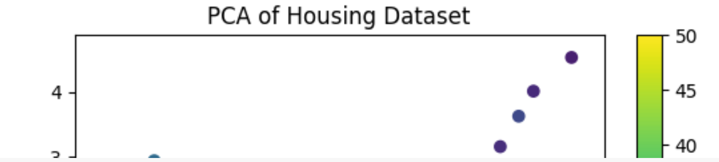
two concerns 'chas' is categorical will try with and without

```
features_withoutchas = features.drop('chas', axis=1)
x = StandardScaler().fit_transform(features_withoutchas)
x=pd.DataFrame(x)
x=pd.concat([x,features["chas"]],axis=1)
x=np.array(x)

pca = PCA(.95)

principalComponents = pca.fit_transform(x)
```

```
pca.explained_variance_ratio_
```

```
array([0.5077823 , 0.11137644, 0.09813515, 0.06923388, 0.05511658,
       0.04456185, 0.03279204, 0.02296884, 0.01818564])
```

```
plt.scatter(principalComponents[:, 0], principalComponents[:, 1], c=boston_housing['medv'], cmap='viridis')
plt.colorbar(label='medv')
plt.xlabel('Principal Component 1')
plt.ylabel('Principal Component 2')
plt.title('PCA of Housing Dataset')
plt.show()
```




```
x_nochas = StandardScaler().fit_transform(features_withoutchas)

principalComponents_nochas = pca.fit_transform(x_nochas)
```

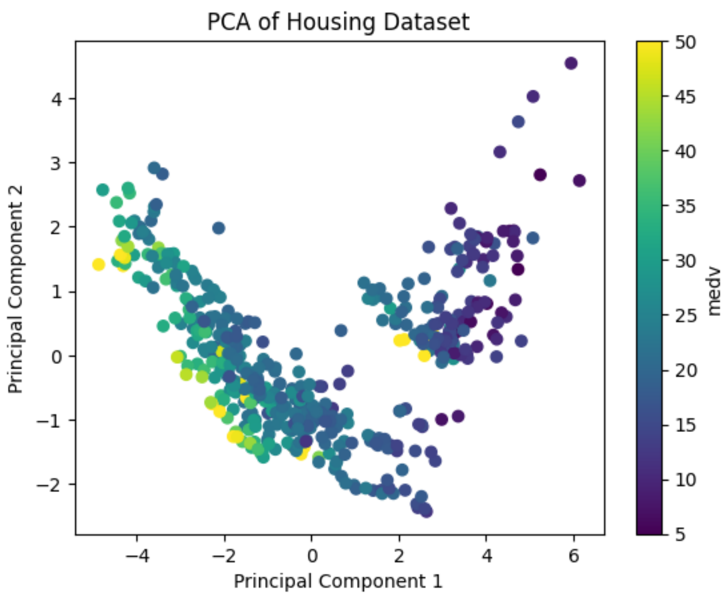
```
x_nochas.shape
```

    (506, 12)

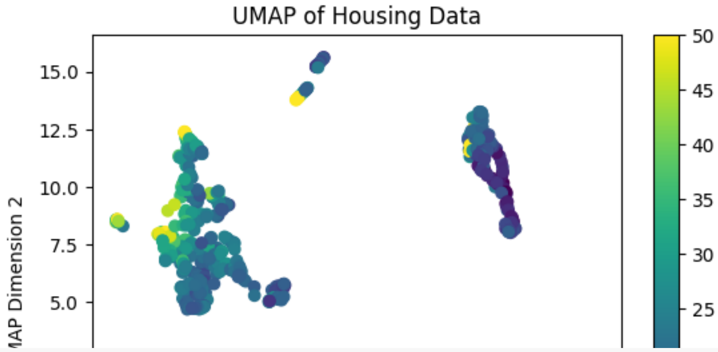
```
pca.explained_variance_ratio_
```

    array([0.5105062, 0.1118566])

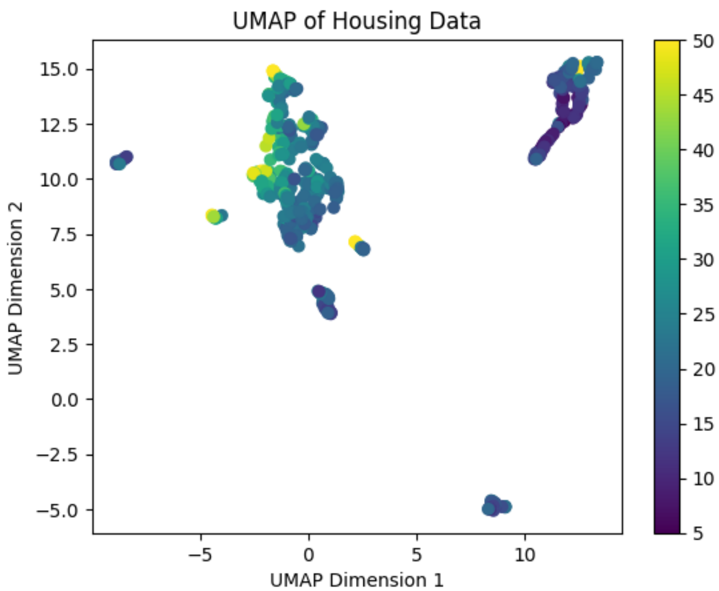
```
plt.scatter(principalComponents_nochas[:, 0], principalComponents_nochas[:, 1], c=boston_housing['medv'], cmap='viridis')
plt.colorbar(label='medv')
plt.xlabel('Principal Component 1')
plt.ylabel('Principal Component 2')
plt.title('PCA of Housing Dataset')
plt.show()
```



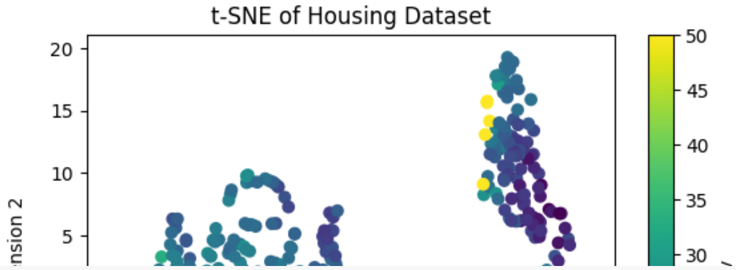

```
umap_model = umap.UMAP(n_components=2)
housing_umap = umap_model.fit_transform(x)

# Scatter plot after UMAP
plt.scatter(housing_umap[:, 0], housing_umap[:, 1], c=boston_housing["medv"], cmap="viridis")
plt.xlabel("UMAP Dimension 1")
plt.ylabel("UMAP Dimension 2")
plt.title("UMAP of Housing Data")
plt.colorbar()
plt.show()
```



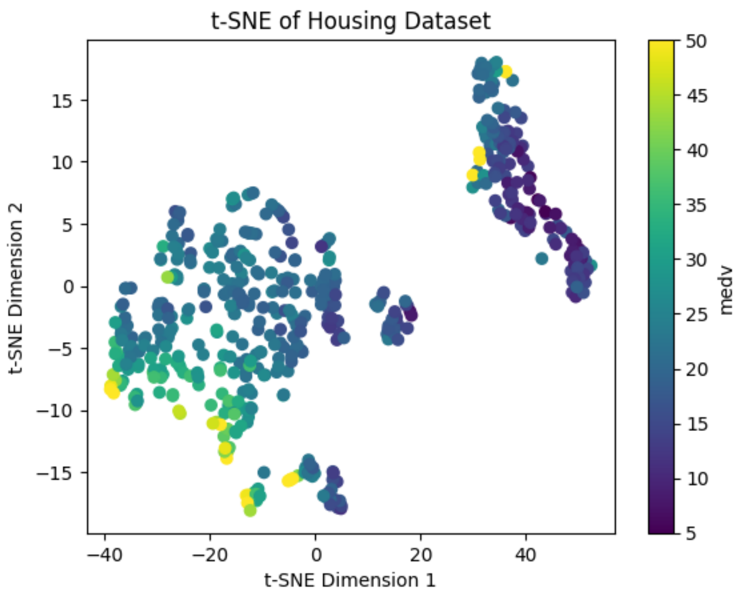

```
housing_umap_nochas = umap_model.fit_transform(x_nochas)

# Scatter plot after UMAP
plt.scatter(housing_umap_nochas[:, 0], housing_umap_nochas[:, 1], c=boston_housing["medv"], cmap="viridis")
plt.xlabel("UMAP Dimension 1")
plt.ylabel("UMAP Dimension 2")
plt.title("UMAP of Housing Data")
plt.colorbar()
plt.show()
```




```
# Applying t-SNE
tsne = TSNE(n_components=2,random_state=42)
housing_tsne = tsne.fit_transform(x)

# Plotting the t-SNE results
plt.scatter(housing_tsne[:, 0], housing_tsne[:, 1], c=boston_housing['medv'], cmap='viridis')
plt.colorbar(label='medv')
plt.xlabel('t-SNE Dimension 1')
plt.ylabel('t-SNE Dimension 2')
plt.title('t-SNE of Housing Dataset')
plt.show()
```




```
housing_tsne_nochas = tsne.fit_transform(x_nochas)

# Plotting the t-SNE results
plt.scatter(housing_tsne_nochas[:, 0], housing_tsne_nochas[:, 1], c=boston_housing['medv'], cmap='viridis')
plt.colorbar(label='medv')
plt.xlabel('t-SNE Dimension 1')
plt.ylabel('t-SNE Dimension 2')
plt.title('t-SNE of Housing Dataset')
plt.show()
```



Tried each dimensionality reduction technique (including and excluding 'chas' feature) as a visualization tool and noticed that the first dimension (on x-axis) separates lower median value houses and higher median value houses quite well. Overall, it is still hard to interpret as is the case with dimensionality reduction techniques. Moving on, dataset will **NO LONGER USE * 'chas' because of the overall low variance (info value) and 'rad' (high correlation with tax).

Cross-Validation Strategy

Our group has concluded that the best cross-validation strategy would be to use some permutation of kfold cv. This is the case because our dataset is small, and we have decided to drop some features. kfold will allow us to avoid underfitting while mainting a balance of variance. Since our dataset is not a classification dataset, it was hard to justify stratified cv as a good strategy and instead we went for a simpler kfold.

Model Building and Experimentation

Start with simple linear model

```
features=features.drop(['chas','rad'],axis=1)
```

```
X_train, X_test, y_train, y_test = train_test_split(features, target_variable, test_size=0.2, random_state=42)
```

```
X_train, X_test, y_train, y_test = train_test_split(features, target_variable, test_size=0.2, random_state=42)

linear_model_base=LinearRegression().fit(X_train,y_train)
train_pred=linear_model_base.predict(X_train)


mse = metrics.mean_squared_error(y_train, train_pred)
rmse = np.sqrt(mse) #mse**(0.5)
r2 = metrics.r2_score(y_train,train_pred)

print("Train MSE:", mse)
print("Train RMSE:", rmse)
print("Train R-Squared:", r2)
```

```
Train MSE: 22.925500186282807
Train RMSE: 4.788058080922036
Train R-Squared: 0.736104501783255
```

```
## Test Metrics

y_hat=linear_model_base.predict(X_test)
mse = metrics.mean_squared_error(y_test, y_hat)
rmse = np.sqrt(mse) #mse**(0.5)
r2 = metrics.r2_score(y_test,y_hat)
print("Test MSE:", mse)
print("Test RMSE:", rmse)
print("Test R-Squared:", r2)
```

```
Test MSE: 27.304132213791554
Test RMSE: 5.22533560776641
Test R-Squared: 0.6276732082115685
```

Linear model is not the best at house price predicting

```
linear_scores = cross_val_score(linear_model_base, X_train, y_train, scoring="neg_mean_squared_error", cv=10)
print(-linear_scores)
```

```
[14.77055266 18.42757285 29.37484809 44.33827652 25.33293528 30.08698198
 20.01305631 18.81680525 12.64071601 37.98959581]
```

As shown above the cv scores are very inconsistent alluding to the idea that our data is inherently bad

```
## first train simple regressor tree

DTregressor = DecisionTreeRegressor(random_state=42).fit(X_train,y_train)
```

```
DTregressor.score(X_train,y_train) #R^2 of train test
```

```
1.0
```

```
DTregressor.score(X_test,y_test) #R^2 of test set
```

```
0.8354849663551922
```

```
print("MSE DT test: ", metrics.mean_squared_error(y_test,(DTregressor.predict(X_test))))
```

```
MSE DT test:  12.06450980392157
```

```
DTregressor.get_depth()
```

```
19
```

```
DTregressor.get_n_leaves()
```

```
382
```

Decision Tree results in much better r^2

```
clf = GridSearchCV(DecisionTreeRegressor(random_state=42), {
    'max_depth': np.arange(1,20),
    'splitter': ['best','random'],
    'max_leaf_nodes': np.arange(2,400)
}, cv=10, return_train_score=False)
clf.fit(X_train, y_train)
DTgridcvresults=pd.DataFrame(clf.cv_results_)
DTgridcvresults
```

	mean_fit_time	std_fit_time	mean_score_time	std_score_time	param_max_depth	pa
0	0.002946	0.000519	0.001757	0.000341	1	
1	0.002534	0.000319	0.001687	0.000381	1	
2	0.003098	0.000671	0.001698	0.000187	1	
3	0.002428	0.000147	0.001600	0.000099	1	
4	0.002626	0.000120	0.001567	0.000080	1	
...	...	...	...	...	...	
15119	0.003694	0.000245	0.001813	0.000127	19	
15120	0.005900	0.000662	0.002114	0.000238	19	
15121	0.003580	0.000240	0.001819	0.000237	19	
15122	0.006115	0.000437	0.002322	0.000196	19	
15123	0.003555	0.000117	0.001748	0.000048	19	

15124 rows × 21 columns

```
print(clf.best_estimator_,clf.best_score_,)
```

```
DecisionTreeRegressor(max_depth=7, max_leaf_nodes=30, random_state=42,
                      splitter='random') 0.7109387186004412
```

```
bestDTpred=clf.predict(X_test)

print("Grid CV R^2 DT Test: ", metrics.r2_score(y_test,bestDTpred))
print("Grid CV MSE DT Test: ", metrics.mean_squared_error(y_test,bestDTpred))
```

```
Grid CV R^2 DT Test:  0.7841876323568392
Grid CV MSE DT Test:  15.82633737205945
```

The grid search interestingly gave worse metric results but also created a simpler model. 7 depth, with 30 max leafs is much simpler than a tree with depth of 19 with 382 leaves

```
clf.best_estimator_.feature_importances_
```

```
array([0.01484579, 0.00950113, 0.01994083, 0.02944437, 0.39039124,
       0.02280508, 0.01479665, 0.0353148 , 0.00894195, 0.01805693,
       0.43596124])
```

```
pd.DataFrame({"Features":X_train.columns, "Importance":clf.best_estimator_.feature_importances_})
```

	Features	Importance
0	crim	0.014846
1	zn	0.009501
2	indus	0.019941
3	nox	0.029444
4	rm	0.390391
5	age	0.022805
6	dis	0.014797
7	tax	0.035315
8	ptratio	0.008942
9	b	0.018057
10	lstat	0.435961

The feature importances show that rm and lstat are above and beyound more important than any other feature.

Model Improvement Proposals

Feature Selection

Remove worst feature indicated from above model and retrain (with cv) on the same Decision Tree Regressor

```
features_nopratio=features.drop(["ptratio"],axis=1)
```

```
X_trainfs, X_testfs, y_trainfs, y_testfs = train_test_split(features_nopratio, target_variable, test_size=0.2, random_state=42)
```

```
fsDT=DecisionTreeRegressor(random_state=42).fit(X_trainfs,y_trainfs)
fsDT_pred=fsDT.predict(X_testfs)
print("Feature Selected DT Test R^2: ", metrics.r2_score(y_testfs,fsDT_pred))
print("Feature Selected DT Test MSE: ", metrics.mean_squared_error(y_testfs,fsDT_pred))
```

```
Feature Selected DT Test R^2:  0.8726371448806778
Feature Selected DT Test MSE:  9.34
```

```
fsDT.get_depth()
```

```
19
```

```
fsDT.get_n_leaves()
```

```
380
```

```
fsDT.get_params()
```

```
{'ccp_alpha': 0.0,
 'criterion': 'squared_error',
 'max_depth': None,
 'max_features': None,
 'max_leaf_nodes': None,
 'min_impurity_decrease': 0.0,
 'min_samples_leaf': 1,
 'min_samples_split': 2,
 'min_weight_fraction_leaf': 0.0,
 'random_state': 42,
 'splitter': 'best'}
```

```
clf_featureselected = GridSearchCV(DecisionTreeRegressor(random_state=42), {
    'max_depth': np.arange(1,20),
    'splitter': ['best','random'],
    'max_leaf_nodes': np.arange(2,400)
}, cv=10, return_train_score=False)
clf_featureselected.fit(X_trainfs, y_trainfs)
fsDTgridcvresults=pd.DataFrame(clf_featureselected.cv_results_)
fsDTgridcvresults
```

	mean_fit_time	std_fit_time	mean_score_time	std_score_time	param_max_depth	pa
0	0.003584	0.001091	0.002035	0.000516	1	
1	0.002554	0.000134	0.001586	0.000076	1	
2	0.002948	0.000340	0.001672	0.000163	1	
3	0.002820	0.000333	0.001720	0.000153	1	
4	0.003249	0.000285	0.001895	0.000210	1	
...	...	...	...	...	...	
15119	0.005266	0.000116	0.002676	0.000136	19	
15120	0.008056	0.000257	0.003427	0.001193	19	
15121	0.005317	0.000123	0.002811	0.000264	19	
15122	0.008609	0.001572	0.003587	0.001114	19	
15123	0.005770	0.000717	0.002854	0.000202	19	

15124 rows × 21 columns

```
print(clf_featureselected.best_estimator_,clf_featureselected.best_score_,)
```

```
DecisionTreeRegressor(max_depth=11, max_leaf_nodes=31, random_state=42,
                      splitter='random') 0.7463900490732635
```

```
bestDTpredfs=clf_featureselected.predict(X_testfs)

print("Grid CV R^2 DT Test: ", metrics.r2_score(y_testfs,bestDTpredfs))
print("Grid CV MSE DT Test: ", metrics.mean_squared_error(y_testfs,bestDTpredfs))
```

```
Grid CV R^2 DT Test:  0.6096079431798278
Grid CV MSE DT Test:  28.62892644236298
```

We see that grid search cv leads again to a worse model, but this time the metrics are much worse, so the smaller model may not make sense in this case and better to just use default parameters on the DecisionTreeRegressor with the dropped feature.

Increased Model Complexity

The next proposal is to use a more complex model, specifcally an ensemble method of RandomForestRegressor. Use the same dataset as used in the first DTRegressor model and apply same cv approach.

```
rf=RandomForestRegressor(random_state=42).fit(X_test,y_test)
print("train set RF R^2: ", rf.score(X_train,y_train))
print("test set RF R^2: ", rf.score(X_test,y_test))
```

```
train set RF R^2:  0.8146972847430404
test set RF R^2:  0.9693605585307042
```

```
rfpred=rf.predict(X_test)
print("test set RF MSE: ", metrics.mean_squared_error(rfpred,y_test))
```

```
test set RF MSE:  2.2469061568627438
```

Right off the bat, the R^2 scores look great for default random forest regressor, and an exceptionally low MSE

```
rf.get_params()
```

```
{'bootstrap': True,
 'ccp_alpha': 0.0,
 'criterion': 'squared_error',
 'max_depth': None,
 'max_features': 1.0,
 'max_leaf_nodes': None,
 'max_samples': None,
 'min_impurity_decrease': 0.0,
 'min_samples_leaf': 1,
 'min_samples_split': 2,
 'min_weight_fraction_leaf': 0.0,
 'n_estimators': 100,
 'n_jobs': None,
 'oob_score': False,
 'random_state': 42,
 'verbose': 0,
 'warm_start': False}
```

```
clf_rf = GridSearchCV(RandomForestRegressor(random_state=42), {
    'max_depth': np.arange(1,20),
    'n_estimators': np.arange(50,150),
}, cv=10, return_train_score=False)
clf_rf.fit(X_train, y_train)
RFgridcvresults=pd.DataFrame(clf_rf.cv_results_)
RFgridcvresults
```

	mean_fit_time	std_fit_time	mean_score_time	std_score_time	param_max_depth	param_n_estimators	params	split0_test_score
0	0.079775	0.007883	0.003804	0.000236	1	50	{'max_depth': 1, 'n_estimators': 50}	0.625816
1	0.077003	0.002804	0.004029	0.000781	1	51	{'max_depth': 1, 'n_estimators': 51}	0.627426
2	0.080201	0.004814	0.004173	0.000858	1	52	{'max_depth': 1, 'n_estimators': 52}	0.628685
3	0.080021	0.002920	0.003809	0.000107	1	53	{'max_depth': 1, 'n_estimators':	0.629867

```
print(clf_rf.best_estimator_,clf_rf.best_score_,)
```

```
RandomForestRegressor(max_depth=13, n_estimators=146, random_state=42) 0.8236368259793689
```

```
bestRFpred=clf_rf.predict(X_test)

print("Grid CV R^2 Rf Test: ", metrics.r2_score(y_test,bestRFpred))
print("Grid CV MSE RF Test: ", metrics.mean_squared_error(y_test,bestRFpred))
```

```
Grid CV R^2 Rf Test:  0.8758813347061408
Grid CV MSE RF Test:  9.102091286808575
```

Of all experiments, the best model performance was the default Random Forest Regressor with a R^2 of .969 and MSE of 2.247

	mean_fit_time	std_fit_time	mean_score_time	std_score_time	param_max_depth	param_n_estimators	params	split0_test_score
							146}	
1897	0.484681	0.021206	0.008456	0.001287	19	147	{'max_depth': 19, 'n_estimators': 147}	0.831395
1898	0.659911	0.131709	0.011787	0.003024	19	148	{'max_depth': 19, 'n_estimators': 148}	0.831926
1899	0.480904	0.016309	0.008330	0.000449	19	149	{'max_depth': 19, 'n_estimators':	0.831563