Homework 4

In [1]:
```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
```

1. Principal Component Analysis

1. The principal components are the directions in which data points are correlated. In this sense, the computation of the principal components requires the data to be standardized so that we are actually seeing how much data correlates depending on the feature. If the data were not standardized, we would be seeing the correlation of units, not the data itself. If we didn't standardize the data, we could change the units along a given feature and thus the variance along that direction would scale accordingly. This would essentially give us a useless metric, since we could arbitrarily choose the principal component corresponding to largest singular value, obtained by arbitrarily scaling that direction.

2. The loadings are the components of the unit vector that points in the direction of the highest variability in the data. If we pretend the data is normally distributed, then the image of the euclidean ball under the covariance matrix is an ellipsoid, and the longest axis, normalized, would be the first principal component, and the loadings would be its components. The scores for each sample are the inner product with the principal component, i.e. how much does this sample vary along this axis.

3. Code

In [8]:
```
college_df = pd.read_csv('data/College.csv',  true_values=['Yes'], false_values=['N
college_df.head()
```

Out[8]:

	Private	Apps	Accept	Enroll	Top10perc	Top25perc	F.Undergrad	P.Undergrad	Outsta
0	True	1660	1232	721	23	52	2885	537	74
1	True	2186	1924	512	16	29	2683	1227	122
2	True	1428	1097	336	22	50	1036	99	112
3	True	417	349	137	60	89	510	63	129
4	True	193	146	55	16	44	249	869	75

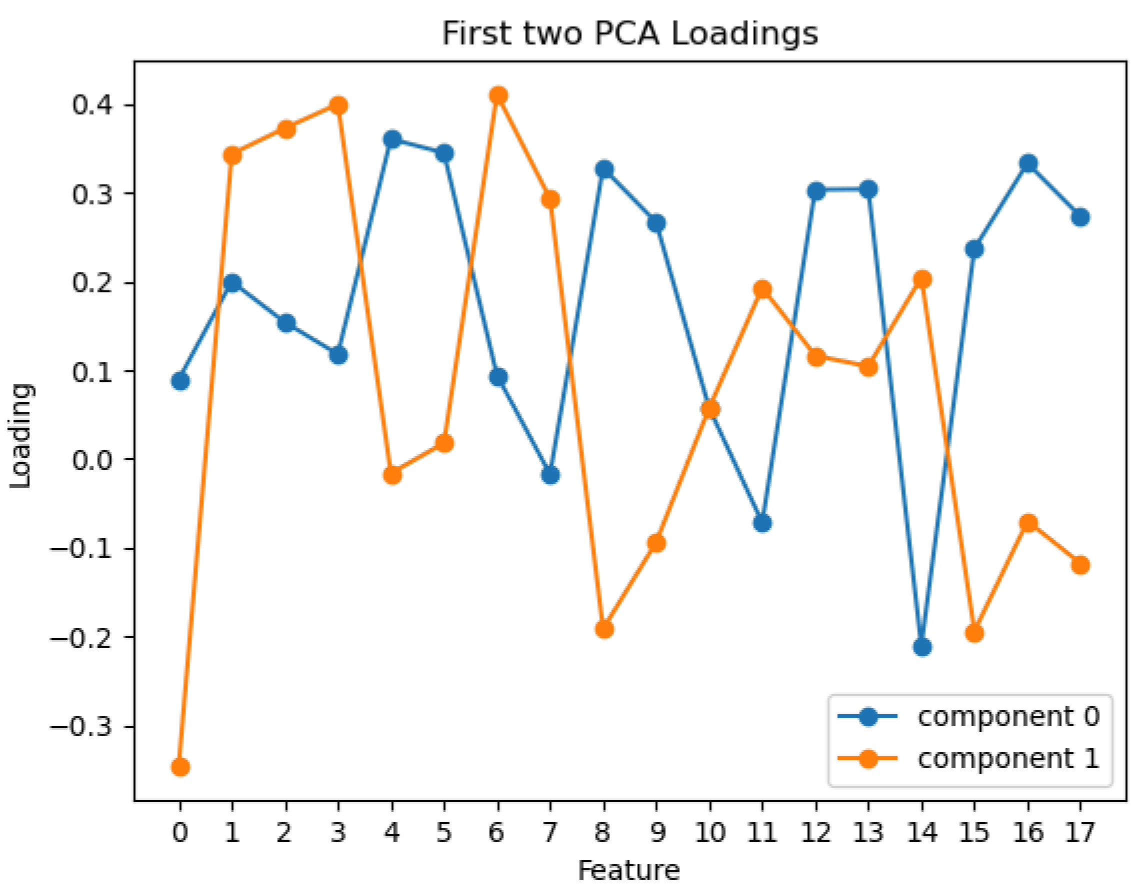
4. Code

In [9]:
```
# standardize columns
college_df = (college_df - college_df.mean()) / college_df.std()
```

In [19]:
```
from sklearn.decomposition import PCA
# fit PCA
pca = PCA(n_components=2)
pca.fit(college_df)
```

Out[19]:
```
PCA
PCA(n_components=2)
```

In [22]:
```
# plot loadings
plt.figure
for i, component in enumerate(pca.components_):
    plt.plot(component, marker = 'o', label = f'component {i}')
plt.legend()
plt.xlabel('Feature')
plt.xticks(np.arange(len(college_df.columns)))
plt.ylabel('Loading')
plt.title('First two PCA Loadings')
plt.show()
```



In [ ]:
```python
# where do variables load
factor_loadings_ids = np.argmax(np.abs(pca.components_), axis=0) # for each feature
factor_loadings = pd.DataFrame({'feature': college_df.columns, 'factor': factor_loa
print(factor_loadings)
```

```
        feature  factor
0       Private       1
1          Apps       1
2        Accept       1
3        Enroll       1
4      Top10perc      0
5      Top25perc      0
6    F.Undergrad      1
7    P.Undergrad      1
8      Outstate       0
9     Room.Board      0
10        Books       1
11     Personal       1
12          PhD       0
13     Terminal       0
14     S.F.Ratio      0
15   perc.alumni      0
16       Expend       0
17    Grad.Rate       0
```

The 5 variables that load more into the first factor are Private, number of applications, the number of students accepted, the number of students enrolled, and number of enrolled fulltime and part time undergrads. These obviously vary with each other, since varying any of these coordinates will very likely vary the others.

The 5 variables that load more onto the second factor are the percent of new students from the top 10/25% of their high school class, out of state tuition, room and board costs, percent of the faculty with PhDs/Terminal degrees, and the student to faculty ratio. These make slightly more sense, since better schools with more PhDs and lower student to faculty ration will generally correlate with high tuition, room and board, and also more students that were the top of their class.

5. Code

In [38]:
```python
sorted_scores_id = np.argsort(np.abs(pca.transform(college_df)), axis = 0)
score_sorted_college_df = pd.DataFrame(
    {'Factor 1': sorted_scores_id[:, 0],
     'Factor 2': sorted_scores_id[:, 1]}
    )
print(f"The top 5 samples sorted by scores in the first factor are:\n {college_df.i
print(f"The bottom 5 samples sorted by scores in the first factor are:\n {college_d
print(f"The top 5 samples sorted by scores in the second factor are:\n {college_df.
print(f"The bottom 5 samples sorted by scores in the second factor are:\n {college_
```

```
The top 5 samples sorted by scores in the first factor are:
      Private      Apps    Accept    Enroll  Top10perc
622 -1.631461  0.149440  0.238339  0.911589   0.478530
209  0.612159 -0.567836 -0.565785 -0.537006   0.081713
603 -1.631461 -0.311260 -0.309575  0.170072  -0.201728
131  0.612159 -0.643542 -0.655948 -0.639247  -0.258416
132  0.612159 -0.660854 -0.677163 -0.708125   0.365154
The bottom 5 samples sorted by scores in the first factor are:
      Private      Apps    Accept    Enroll  Top10perc
158  0.612159  1.443171  0.103706  0.330429   3.369627
174  0.612159  2.787287  0.764630  0.864235   3.539691
250  0.612159  2.806924  0.059645  0.888989   3.539691
775  0.612159  1.990429  0.177142  0.577960   3.823132
284  0.612159  1.413973  0.582264  0.141014   2.689369
The top 5 samples sorted by scores in the second factor are:
      Private      Apps    Accept    Enroll  Top10perc
671  0.612159 -0.179742 -0.121498 -0.260417  -0.258416
221  0.612159  2.096367  0.351757  0.656525   2.462616
578  0.612159  0.364932 -0.211661 -0.478890   0.138401
285 -1.631461 -0.560342 -0.550690 -0.539158  -1.392180
70   0.612159  2.476450  0.497813  0.734013   3.369627
The bottom 5 samples sorted by scores in the second factor are:
      Private       Apps    Accept    Enroll  Top10perc
366 -1.631461   3.904800  5.335205  5.811629  -0.258416
581 -1.631461   2.964280  3.467891  6.039788   1.215476
685 -1.631461   3.036111  3.081536  4.895764   1.158788
461 -1.631461   4.858238  6.823508  5.482305   0.081713
483 -1.631461  11.651166  9.918427  4.025100   0.478530
```

6. The first factor captures the size of the school and the second factor captures the resources and quality of the school.

7. Code

In [40]:
```
scores = pca.transform(college_df)
print(f'The variance of the scores for the prinicpal components are: {scores.var(ax
```

```
The variance of the scores for the prinicpal components are: [5.45290636 5.04957944]
```

The variance of the first score is higher than the variance of the second score. This makes sense, since the first factor should explain most of the variability in the data, and the second factor should explain any of the variability that the second score does not.

2. Matrix Completion

1. A simple choice of vectors $(a_i)_{i=1}^N$ and $(b_j)_{j=1}^p$ is to let $a_i = 1$, for all i, and let $b_j = \overline{x}_j$. This implementation of the baseline approach is good because we can get interpretation how we are treating the samples in our imputation. In this example, we

assume that all of samples are drawn from the same distribution and each of our columns is also independent. This approach makes explicit this assumption.

2. In our baseline, we assumed that all of the samples were drawn from the same distribution. However, if we change the weighting, i.e. change a_i from 1, we can parameterize our distribution to account for this difference. In other words, we now assume that each sample first draws a parameter from some distribution and then based on that parameter, draws the actual sample from a distribution parameterized by that parameter. This leads us to want to impute the value in a biased way. Given our missing sample in column j, we want to fill that missing sample from a weighted mean that weighs samples higher, if their column entries, in other columns, are more similar to the other column entries in our missing sample.

3. PCA leads to fill in the values using a weighted mean based on samples that are correlated with our missing sample. In fact, if we imputed the matrix with the baseline, and then ran PCA on this imputed matrix, the a_i's would exactly be a weighted average biased towards samples the covary with the missing sample.

4. Code

In [50]:
```
# standardize boston data
boston_raw = pd.read_csv('data/Boston.csv')
boston_std = (boston_raw - boston_raw.mean()) / boston_raw.std()
```

In [ ]:
```
# make missing function
missing_cols = ['crim', 'indus', 'age', 'dis', 'tax', 'medv']
def make_missing(row):
    rng = np.random.default_rng()
    for col in missing_cols:
        if rng.random() < 0.2:
            row[col] = np.nan
    return row
```

In [ ]:
```
# apply make_missing function
boston_missing = boston_std.apply(make_missing, axis=1)
```

In [45]:
```
# save missing data
boston_missing.to_csv('Boston_missing.csv', index=False)
```

5. We implement the algorithm detailed in the book.

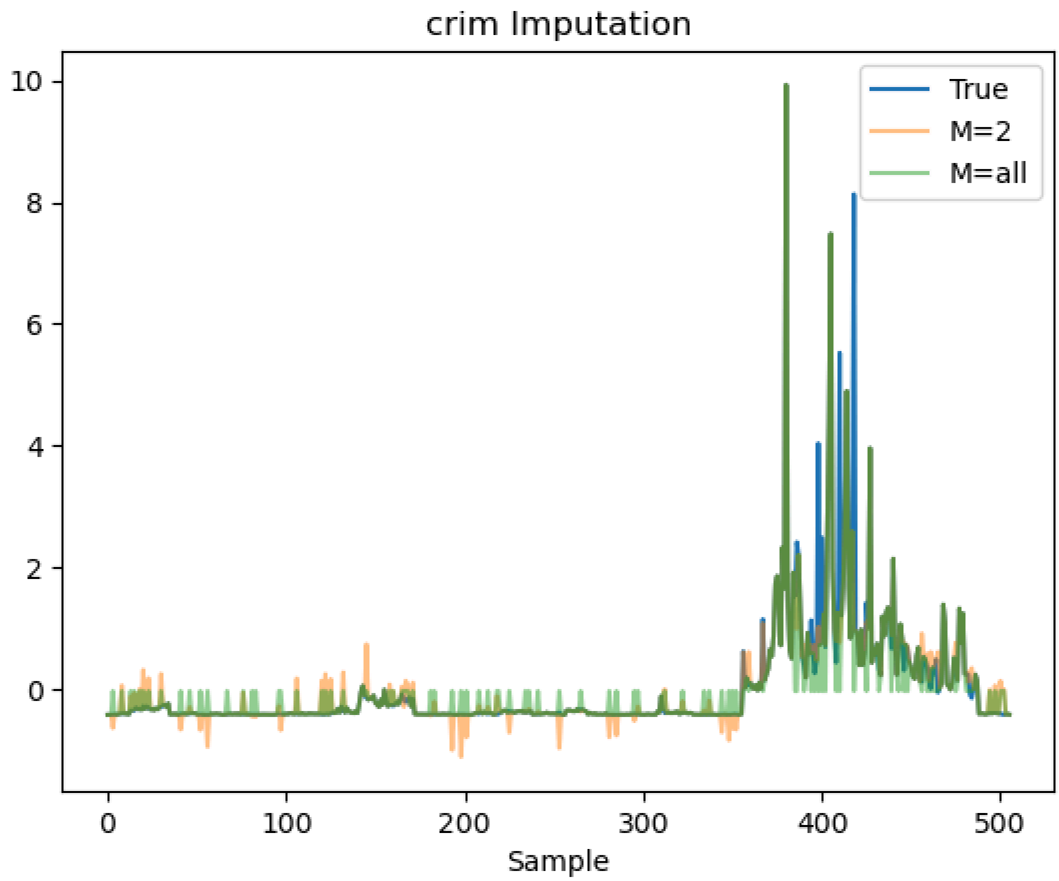
In [83]:
```
# helper function
def low_rank_approximation(X, M = 2):
    pca = PCA(n_components=M)
    return pd.DataFrame(pca.inverse_transform(pca.fit_transform(X)), columns=X.colu
```

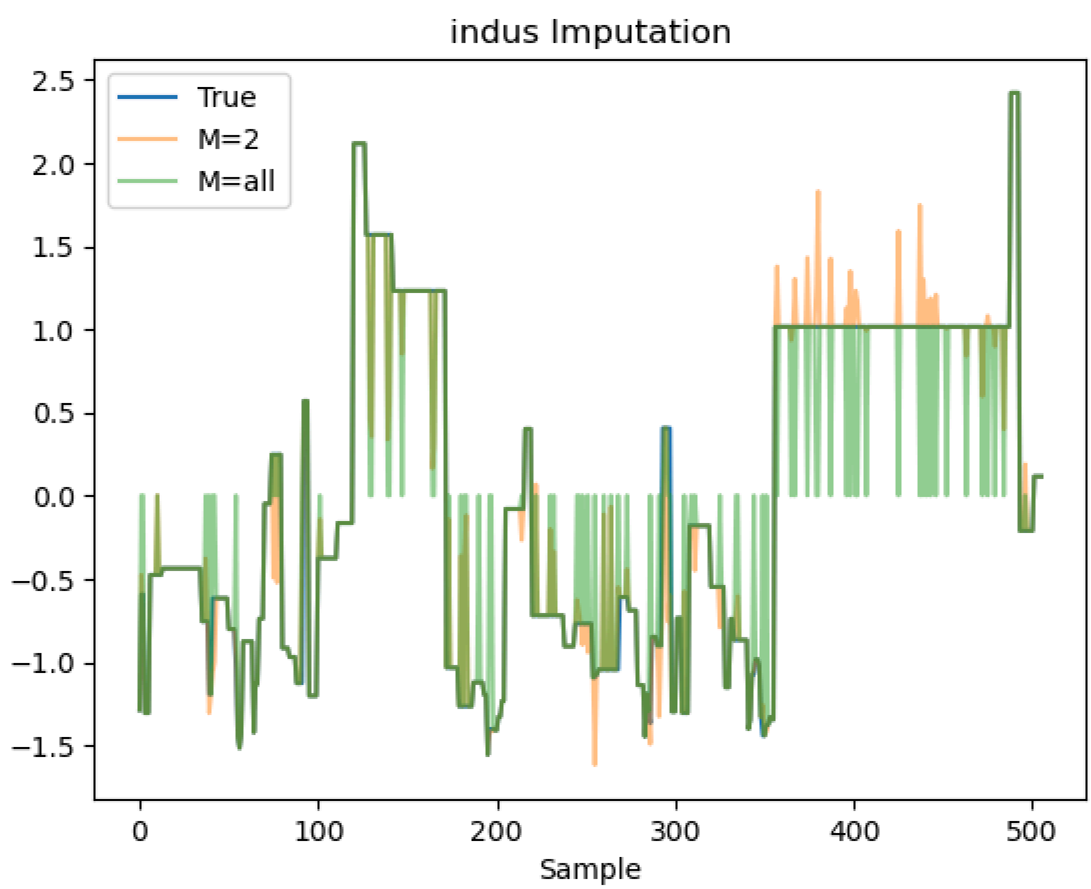
In [86]:
```
def matrix_completion(X: pd.DataFrame, M = 2, thresh = 1e-5):
    # load data
    is_missing = np.isnan(X)
    X_hat = X.copy()
    X_hat = X_hat.fillna(X_hat.mean())

    # init variables
    mss_0 = np.mean(X[~is_missing] ** 2)
    mss_old = np.mean(X_hat[~is_missing] ** 2)
    rel_err = 1

    # loop until done
    while rel_err > thresh:
        X_low_rank = low_rank_approximation(X_hat, M= M)
        X_hat[is_missing] = X_low_rank[is_missing]
        mss = np.mean((X - X_low_rank)[~is_missing] ** 2)
        rel_err = np.abs(mss - mss_old) / mss_0
        mss_old = mss
    return X_hat
```

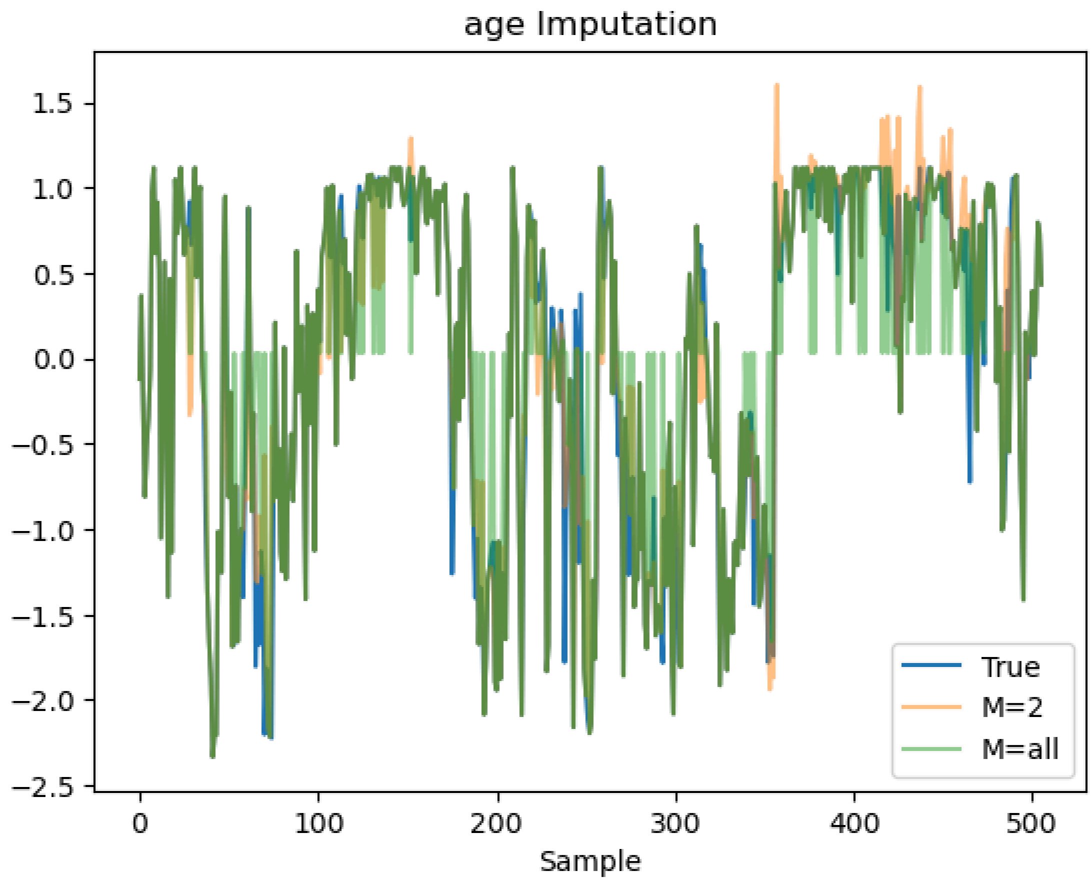
In [89]:
```
boston_missing = pd.read_csv('Boston_missing.csv')
boston_hat_2 = matrix_completion(boston_missing, M = 2)
boston_hat_all = matrix_completion(boston_missing, M=None)
boston_hat_10 = matrix_completion(boston_missing, M=10)
boston_hat_12 = matrix_completion(boston_missing, M=12)
```

6. Code

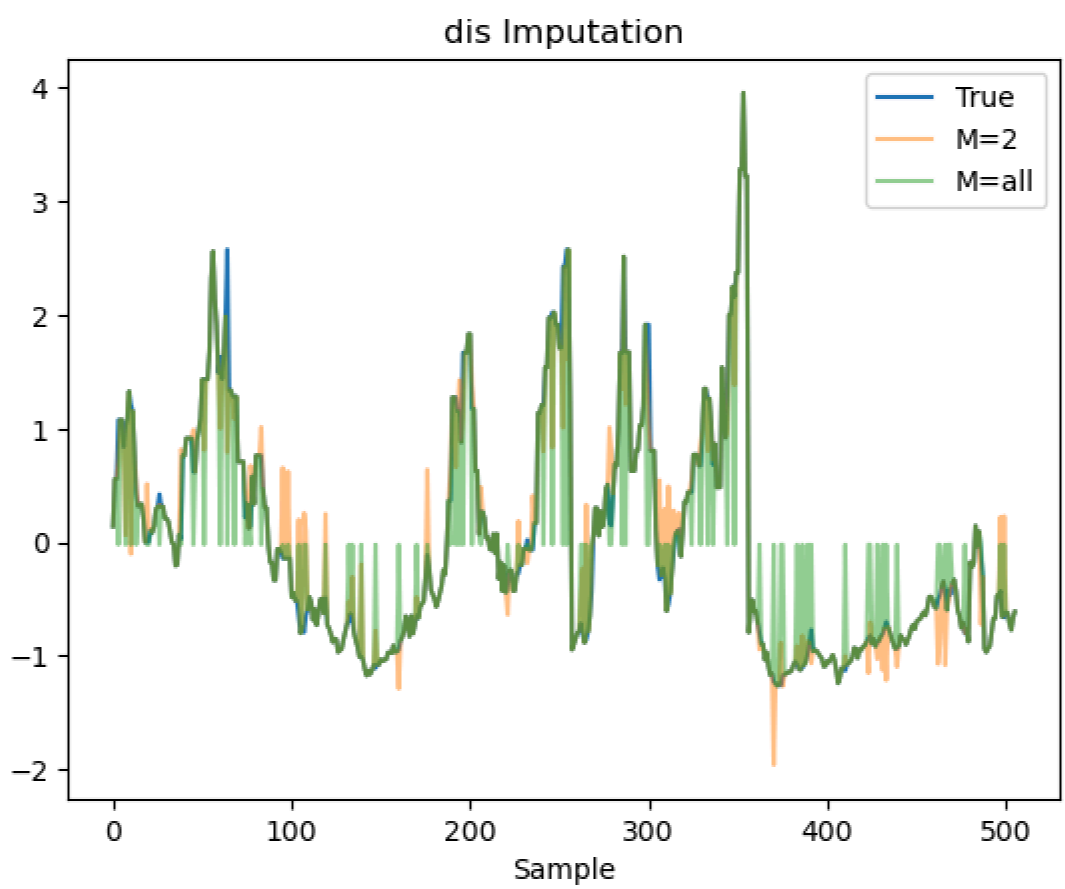
In [93]:
```
for col in missing_cols:
    plt.figure()
    plt.plot(boston_std[col], label='True')
    plt.plot(boston_hat_2[col], label='M=2', alpha=0.5)
    # plt.plot(boston_hat_10[col], label='M=10')
    # plt.plot(boston_hat_12[col], label='M=12')
    plt.plot(boston_hat_all[col], label='M=all', alpha=0.5)
    plt.xlabel('Sample')
    plt.title(f'{col} Imputation')
    plt.legend()
    plt.show()
```


crim Imputation
True
M=2
M=all
Sample

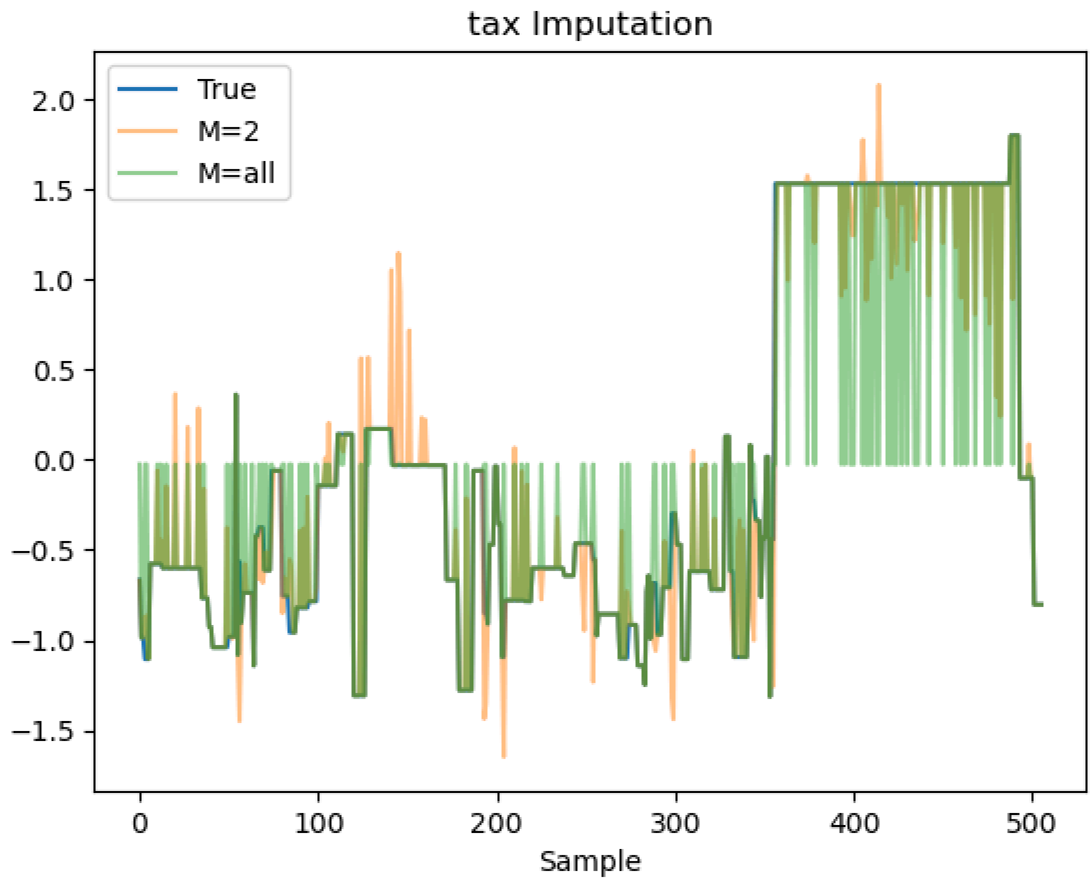

indus Imputation
True
M=2
M=all
Sample

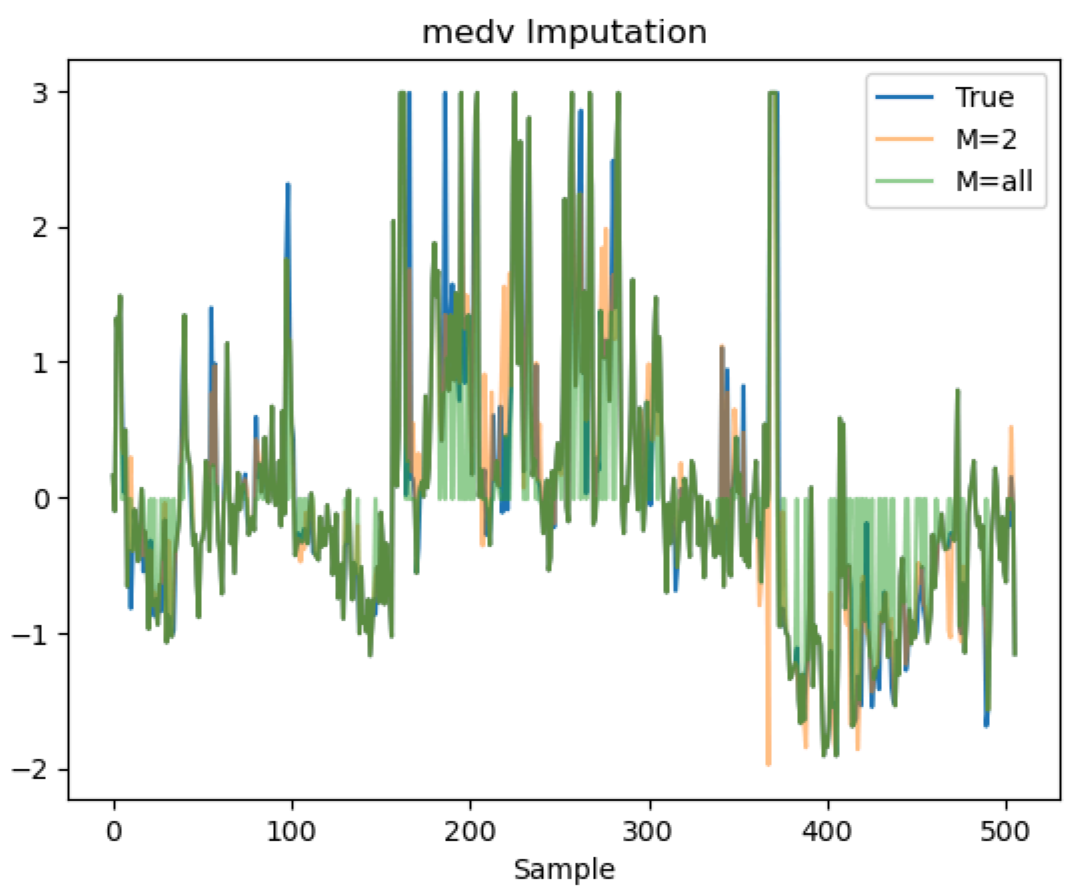


age Imputation
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−2.0
−2.5
0
100
200
300
400
500
Sample
True
M=2
M=all



dis Imputation
4
3
2
1
0
−1
−2
0
100
200
300
400
500
Sample
True
M=2
M=all


tax Imputation
True
M=2
M=all
2.0
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
0
100
200
300
400
500
Sample


medv Imputation
True
M=2
M=all
3
2
1
0
−1
−2
0
100
200
300
400
500
Sample

7. Code

In [98]:
```
mse_2 = np.mean((boston_std - boston_hat_2).loc[:, missing_cols] ** 2, axis = 0)
mse_10 = np.mean((boston_std - boston_hat_10).loc[:, missing_cols] ** 2, axis = 0)
mse_12 = np.mean((boston_std - boston_hat_12).loc[:, missing_cols] ** 2, axis = 0)
mse_all = np.mean((boston_std - boston_hat_all).loc[:, missing_cols] ** 2, axis = 0
print(f'The mean squared error for the M=2 imputation is: \n{mse_2}')
print(f'The mean squared error for the M=10 imputation is: \n{mse_10}')
print(f'The mean squared error for the M=12 imputation is: \n{mse_12}')
print(f'The mean squared error for the M=all imputation is: \n{mse_all}')
```

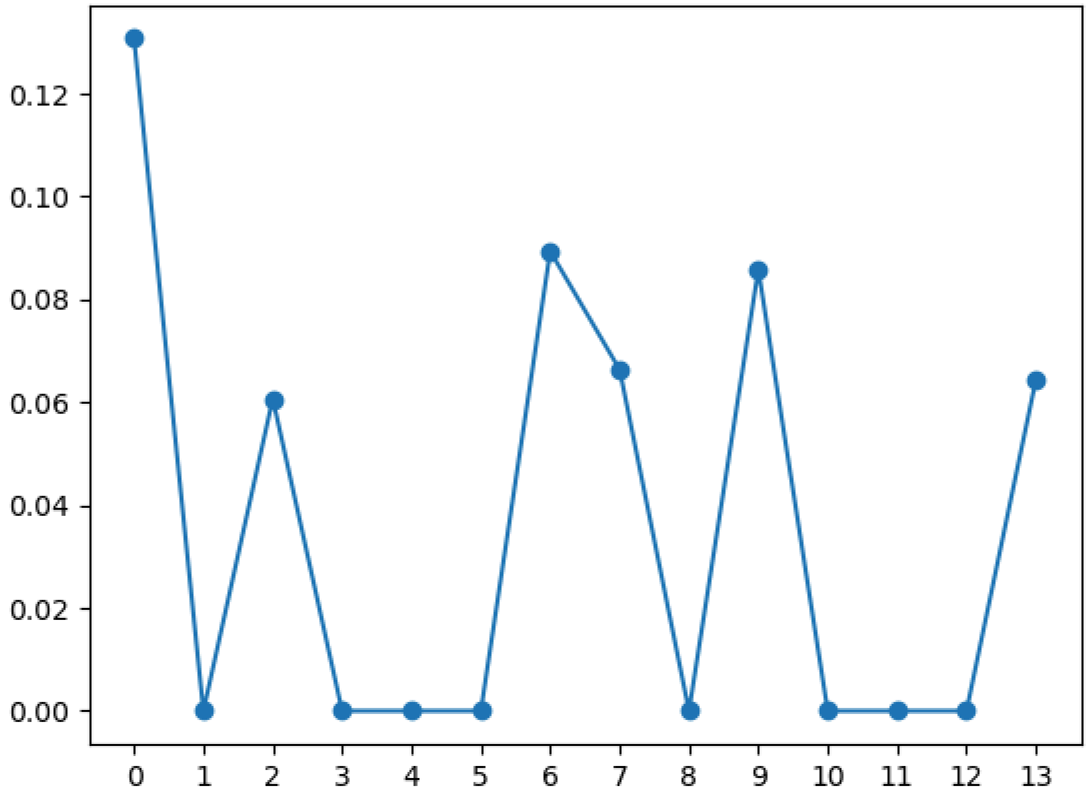
```
The mean squared error for the M=2 imputation is:
crim     0.196336
indus    0.049789
age      0.087832
dis      0.063160
tax      0.072498
medv     0.083243
dtype: float64
The mean squared error for the M=10 imputation is:
crim     0.677130
indus    0.149693
age      0.123228
dis      0.056028
tax      0.023573
medv     0.140151
dtype: float64
The mean squared error for the M=12 imputation is:
crim     0.530006
indus    0.239432
age      0.367849
dis      0.118153
tax      0.036710
medv     0.192410
dtype: float64
The mean squared error for the M=all imputation is:
crim     0.286906
indus    0.152429
age      0.203709
dis      0.181703
tax      0.235802
medv     0.254317
dtype: float64
```

8. Code

In [ ]:
```
def eval_completion(X, cols = ['crim', 'indus', 'age', 'dis', 'tax', 'medv'], M = 2
    X_missing = X.apply(make_missing, axis = 1)
    X_hat = matrix_completion(X_missing, M = M)
    mse = np.mean((X - X_hat).loc[:, cols] ** 2, axis = 0)
    return mse
```

In [102…

```
plt.figure()
mses = []
for col in boston_std.columns:
    mse = eval_completion(boston_std, cols=[col])
    mses.append(mse)
plt.plot(mses, marker='o')
plt.xticks(np.arange(len(boston_std.columns)))
plt.show()
```



In [105…

```
feature_idxs = [1, 3, 4, 5, 8, 10, 11, 12]
feature_subset = boston_std.columns[feature_idxs]
print(f"The mean squared error for the M=2 imputation for the proposed feature subs
```

```
The mean squared error for the M=2 imputation for the proposed feature subset is:
zn          0.0
chas        0.0
nox         0.0
rm          0.0
rad         0.0
ptratio     0.0
black       0.0
lstat       0.0
dtype: float64
```

The proposed subset is the variables:

- zn: proportion of residential land zoned for lots over 25,000 sq.ft.
- chas: Charles River dummy variable (= 1 if tract bounds river; 0 otherwise).

- nox: nitrogen oxides concentration (parts per 10 million).
- rm: average number of rooms per dwelling.
- rad: index of accessibility to radial highways.
- ptratio: pupil-teacher ratio by town.
- lstat: lower status of the population (percent).

A potential reason why the matrix completion works perfectly for these is if there are variables that correlate perfectly with the variable that needs to be replaced.