# CS536 Machine Learning - Final Project Report

Shang-Hao Huang (sh1384)
Zhaoyang Xia (zx149)

Fall 2020

For the final project, given the data set of the form $x_1$, $x_2$, ..., $x_m$, and we want to construct a system for predicting or interpolating missing features (frequently given as empty or NA) from the present features in new records. Instead of singling out a single feature or factor for prediction / regression / classification, any feature of $X$ might be missing and needs to be predicted or reconstructed from the features that are present - and the features that are present may vary from instance to instance.

## 1 The Data

The dataset we choose for this project is House Sales in King County, USA, which was published/released under CC0*: Public Domain and can be obtained from Kaggle. This dataset contains house sale prices for King County, including Seattle, between May 2014 and May 2015. The dataset includes some basic characters of the house like the square feet and the number of different types of rooms. The original dataset consisted 21 variables and 21613 observations, and the description and the type of each variable is shown in Table 1.

## 2 Requirements

All the codes regarding this project are presented in the attached Jupyter Notebook file and the language used is Python. All the machine learning algorithms used in this project were implemented on our own and only basic libraries such as numpy, scipy, pandas, were used. Note that we′ve only used the “LabelEncoder” function from “sklearn.preprocessing” to encode the data.

### 2.1 Describe your Model

#### 2.1.1 Data Representation

Before constructing the model, it’s essential to examine and clean the data. The data cleansing process is as follows:

1. Check if there’s any NaN in the original data and drop the whole record that contains it if any was found: we found nothing. This is because we want all data to have no missing values so that we can have answer to each missing value in order to measure the performance of our data interpolation model.
2. Drop ‘id’ column as it is not a feature, and hence should be excluded from the analysis.
3. Parse ‘date’ column into three new columns (‘year’, ‘month’, and ‘day’) and drop the ‘date’ column. All these columns are categorical feature.
4. Add ’renovated’ column to indicate if the house was renovated in the past (“1” means it was).

Table 1: Variable Types, Names, and Description

| Type | Variable Name | Description |
|---|---|---|
| - | id | unique identifier for each home sold |
| - | date | date of the home sale |
| numerical | price | price of each home sold |
| categorical | bedrooms | number of bedrooms |
| categorical | bathrooms | number of bathrooms (note that ".5" accounts for a room with a toilet but without shower) |
| numerical | sqft_living | square feet of living area |
| numerical | sqft_lot | square feet of the lot |
| categorical | floors | number of floors |
| categorical | waterfront | whether the house has a waterfront ("0" means it doesn't and "1" mean it does) |
| categorical | view | how good the view of the house was (from 0 to 4 shows the view from bad to good) |
| categorical | condition | condition of the house, ranked from 1 to 5 (the higher, the better) |
| categorical | grade | the building's construction quality, ranked from 1 to 13 (the higher, the better) |
| numerical | sqft_above | square feet above ground |
| numerical | sqft_basement | square feet below ground |
| numerical | yr_built | the year of the house was initially built |
| numerical | yr_renovated | the year of the last renovation of the house ('0' if never renovated) |
| categorical | zipcode | 5-digit zipcode of the house |
| numerical | lat | the latitude of the house |
| numerical | long | the longitude of the house |
| numerical | sqft_living15 | average square feet of interior living space for the nearest 15 houses |
| numerical | sqft_lot15 | average square feet of land lot for the nearest 15 houses |

5. Transform data in "yr_renovated" and "yr_built" such that it become the number of years that have passed since 1900.

6. Use "LabelEncoder" from "sklearn.preprocessing" to encode all the categorical features.

After the above processing, the dataset now contains 23 variables. However, after plotting the correlation heatmap (Figure 1) of all the variables, we found that these 'year', 'month', 'and day' were only correlated with each other and that 'renovated' was only correlated to 'yr_renovated', so we dropped all of them and there are 19 variables used for the modeling and experiment. Then, we create a new dataset for this project by randomly removing entries from the original one. Note that Denoising Autoencoder needs uncorrupted data for training. The original data is split into training, validation, and testing dataset using a ratio 6:2:2. All the data in the training data is fully retained while that in the validation and testing set contains missing values. We remove approximately 30% values in validation set and test set randomly. Note that all numerical features are normalized.

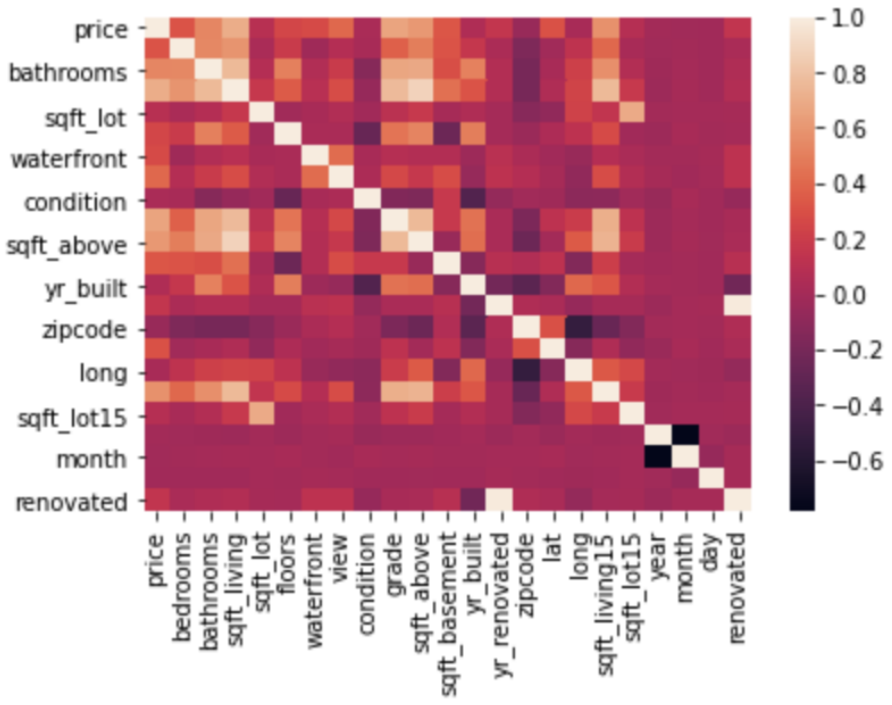


Figure 1: Correlation Heatmap From Training Data

### 2.1.2 Model Pipeline 1 - Models except Denoisng Autoencoder

After cleansing the data, we proposed the architecture of data interpolation Model. The model takes and pre-processes the cleansed data with missing values, performs machine learning algorithms on the data, and then return the data with missing features being interpolated. The steps for our data interpolation process is described below:

1. Sort the columns based on the number of missing values in an increasing order.
2. Find the feature that has the least missing values. Note that, if there is no feature without missing values, then we first use the simplest method to interpolate the missing values for that feature, "Mean" if the feature is numerical, or "Mode" if the feature is categorical.
3. If not, load the regression models if the target feature is numerical and the classification models if the target feature is categorical. Then, use the loaded model to perform imputation for that feature based on the data we have already imputed
4. Perform hold-out validation for each experiment. Note that in each iteration, we will create new data by adding in features we imputed in the previous iterations. The validation data is also created from the new data.
5. Repeat step 2 to 4 for the feature with the second least missing values, the third least missing values, ..., until all the features are imputed.

### 2.1.3 Model Pipeline 2 - Denoisng Autoencoder

Note that for Denoisng Autoencoder, due to the model's structure and usage, we proposed a completely different model pipeline. Instead of doing the data interpolation for each column sequentially, Denoising

Autoencoder can finish the interpolation task in an end-to-end fashion. More details are included in the next section about auto encoder.

## 2.2 Describe your Training Algorithm

The following section shows the machine learning algorithms we've implemented and experimented on the data set. Based on the characteristics of the learning algorithms, the algorithms are first divided into two parts: supervised and unsupervised learning algorithms. For supervised learning part, it can be further divided into regression and classification algorithms, depending on the type of missing values we are trying to fill out.

### 2.2.1 Supervised Learning

- Classification
  1. Mode Model: The mode model is the mean model counterpart for the classification algorithms. It simply predicts the missing value with the most frequent class appears in the remaining data.
  2. Decision Trees Classifier: A decision tree classifier is built through an iterative process that splits the data into branches, and keeps splitting each new branch into smaller groups till a predefined termination criterion is met, noting that target variable is categorical. There are several metric used for splitting the variables, such as information gain and gini impurity, where the latter is implemented for this project. In some cases during experiments, our implementation did not work as expected, thus we did not include the results of this model.
  3. Logistic Regression Classifier: The original logistic regression model is used to model a binary dependent variable, so we implemented a softmax variant that allows to make prediction for variable with multiple categories.
  4. K-Nearest Neighbors (KNN) Classifier: K-Nearest Neighbors model is the last classifier we've implemented. Despite its simplicity, it sometimes can deliver unexpectedly good performance on recognizing patterns, so we would like to experiment it on this project. When predicting a new point, KNN classifier determines its predicted class using the majority class among its K-nearest neighbors. The parameter 'K' used for the experiment is 5.
- Regression
  1. Mean Model: What mean model does is simply predicting by taking the average of the feature we're trying to interpolate. This is the simplest model we've implemented for this project and the purpose of this model is to serve as a baseline benchmark for other algorithms.
  2. Decision Trees Regressor: Similar to the decision tree classifier, the only difference is that the target variable is numerical rather than categorical. As the target variable is different, the metric used here is standard deviation reduction, where the variable that reduces the deviation the most is selected as the splitting node.

     Linear regression is a common and an intuitive technique for linear interpolation as it tries to find a line that best fits to the data at its simplest form. There are three types of regression algorithms implemented: Naive Linear Regression, Ridge Regression, and Lasso Regression.
  3. Naive Linear Regressor: Naive linear regression model solves the problem by finding the weights and bias that minimize the training error, the mean-squared error. The solver implemented here was the closed-form solution with matrix representation.
  4. Ridge Regressor: Ridge regression is a regularized linear regression, which utilizes a penalty term to try to pull the coefficients of insignificant variables toward zero, but not exactly zero. The goal of this method is to prevent the over-fitting problem of naive linear regression model given the

size of training dataset is small. Note that the ridge solver we implemented for this project is the closed-form solution with matrix representation. The regularization constant $\lambda$ is set to 0.1.

5. Lasso Regressor: Lasso regression is another regularized linear regression. The difference between it and the Ridge regression is the regularization term of lasso regression actually set the coefficients of insignificant variables to zero, so it is capable of performing feature selection on the naive linear regression model. In this project, the lasso solved we implemented is the coordinate descent for lasso regression. The regularization constant $\lambda$ is set to 0.05 and the number of iteration is 100.
6. K-Nearest Neighbors Regressor: Similar to KNN classifiers, but with the target variable being numerical. Therefore, when making prediction for a new point, KNN regressor generates the predicted value by taking the average of its K-nearest neighbors target values. The parameter 'K' used for the experiment is 5.

### 2.2.2 Unsupervised Learning

- K-Means Clustering: Though we finished its implementation, due to time constraint, we did not include it in the experiment.

### 2.2.3 Denoising Autoencoder

- Denoising Autoencoder:

  Autoencoder is a type of neural network that is commonly used to learn a latent representation for a dataset. Denoising Autoencoders are a variant designed to reduce the risk of learning the "Identity Function", meaning the output equals the input, and thus making the Autoencoder ineffective. Hence, it is capable of reconstructing corrupted data.

  The input of the dAE (denoising autoencoder) is a record with missing data. We replace the missing value by a default value 0. The output of the model will be the record itself without any missing data. The structure of the neural network is shown below:

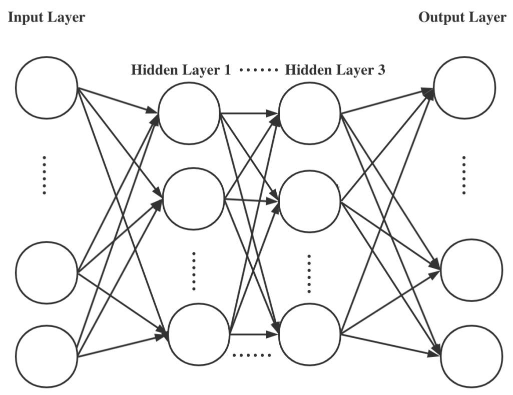


Figure 2: Network Architecture

During our experiment, we found that since our task is to fill in the missing values, we did not need to compress the data by using less units in the hidden layer to find the latent representation. So, we choose three hidden layers with units 30, 40, 20 in our model. We experiment on other parameters as well, for example we experiment with units 15, 10, 15, which will give us a standard denoising autoencoder. However, the performance of the model is slightly worse than the one with more hidden units.

We implement each component of neural network in separate classes from scratch. In this way, we can combine the full connect layer, activation layer together and easily build the neural network in any simple structure. We implement $MSE$ loss function and $ReLu$, $tanh$ activation function. For the optimizer, we apply the mini-batch stochastic gradient descend method.

- Data Centering:

  For Denoising Autoencoder, we regard all the data as numerical data. This is a reasonable approximation and trade off. Looking at the features, many categorical data, such as view, condition, grade, do not have a very strict constraint on their data type. We can also regard bedroom numbers and bathroom numbers as some values in a specific interval, especially after the label encoding. So the majority of features can be considered as numerical data and perform regression task without a big impact. The only feature we might have issues with is zipcode. After label encoding, we set a label with in 60 for each zipcode so we do not have to deal with a value with 5 numbers. Thus we can alleviate this problem.

  Because of the huge difference in our features' scales, we need to perform data centering to help with the numerical stabilization of the neural network. We use the formula below to do the centering for feature $X_i$

$$X_i = \frac{X_i - mean(X_i)}{sd(X_i)}$$

  Note that we calculate the average and standard deviation of features only using training data. And use this mean and standard deviation to do all the centering. In this case we can make sure the information does not leak from test and validation data.

- Experimental Setting

  We use 3 hidden layers with 30, 40, 20 units. The learning rate is $5e^{-4}$. The size of mini-batch is 2000. The loss function we are using is $MSE$ between the output record with the true record.

## 2.3 Describe your Model Validation

### 2.3.1 Models except Denoising Autoencoder

To prevent over-fitting, we used hold-out validation. This part was implemented in our pipeline class "imputation", we split the input data (update in each iteration) into training, validation, and testing set using a ratio 6:2:2. In addition, for each learning algorithm, the prevention of over-fitting work is described below: For linear regression, it is done by introducing the regularization constant $\lambda$ (in both ridge and lasso) and the number of iteration (only in lasso), as mentioned in the description of ridge and lasso regressor. For logistic regression, we used a threshold ($10^{-8}$) to prevent over-fitting by stopping the training process if the updated value of target is less than the threshold. For decision tree regression, it is achieved through terminating the tree-growing process if the number of data points of a branch is less than 3. As for K-Nearest Neighbors, there's no prevention of over-fitting being done.

### 2.3.2 Denoising Autoencoder

We split our data set into training data, validation data and test data. To avoid the over fitting, we check the model's training loss and validation loss every 200 epochs. We set a condition to perform the early stop of training. When we check the validation loss, if we find the validation loss increases, we will stop the training process.

Here is the plot of $MSE$ loss on training data and validation data (Figure 3). As we mentioned before, we also experiment on other parameters, here is the plot of $MSE$ loss when we are using less hidden units (Figure 4).

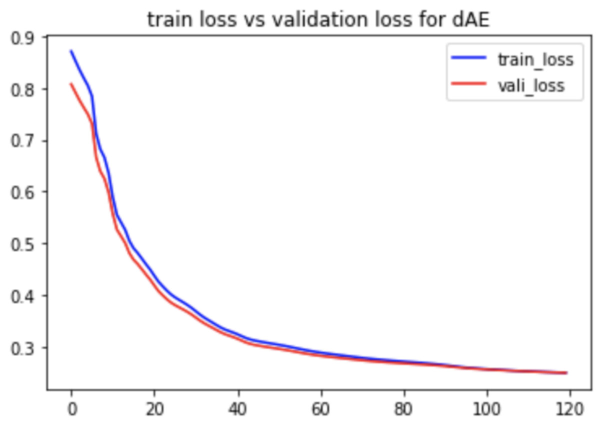


Figure 3: Training Loss vs Validation Loss (more hidden units) The test loss is 0.243

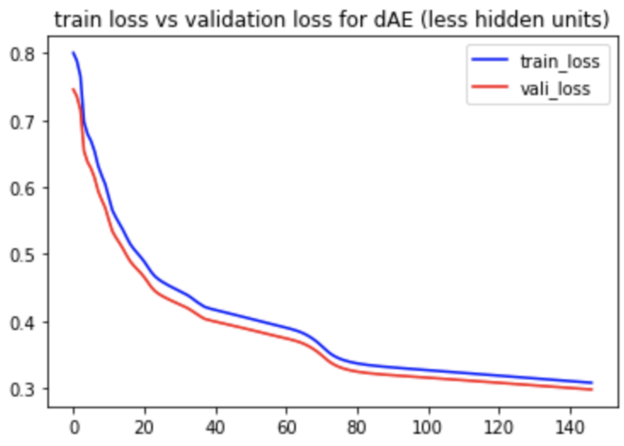


Figure 4: Training Loss vs Validation Loss (less hidden units) The test loss is 0.290

If we do not have enough data, we can use k-fold validation method to fully use our data to train the model. But in our case we think the data is enough to perform training so we only use the method above to do validation.

## 2.4 Evaluation and Analysis of Model

### 2.4.1 Models except Denoising Autoencoder

For models other than Denoising Autoencoder, we use two metrics, 'mean square error' for regression and 'accuracy' for classification, to measure and compare the performance of each model. The results for numerical variables are shown in Figure 5, the resulting $MSE$ ranges from 20 to 60. The best imputed variable overall is "sqft_lot" and the worst one is "sqft_above". Note that the variables are shown from the one with least missing values to the one with the most. Intuitively, because of the iterative nature of our model pipeline, we would expect the variable with more missing values to be better imputed as the number of features in the data used to impute it increases. Unfortunately, we didn't observe the expected trend in the plot. We can see that across almost all features, decision tree regressor has the worst $MSE$ and all three linear regression methods obtained quite similar results. In general, K-nearest has advantage $MSE$ over other models.

For the categorical variables, the result is shown in Figure 6. We can observe the expected trend mentioned previously, as the number of features of the data used to impute increases, the accuracy increases from around 0.4 to almost 1.0. The best imputed variable overall is the "berdooms" and the worst is "grade". In general, K-nearest neighbors has better accuracy over the logistic regression model.

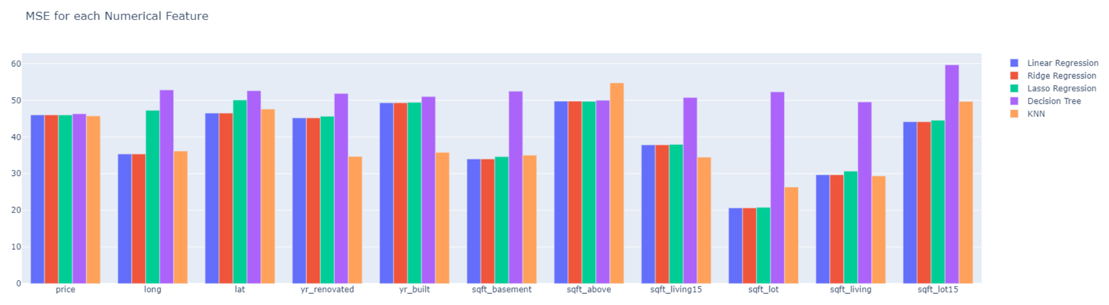


Figure 5: MSE for each Numerical Feature

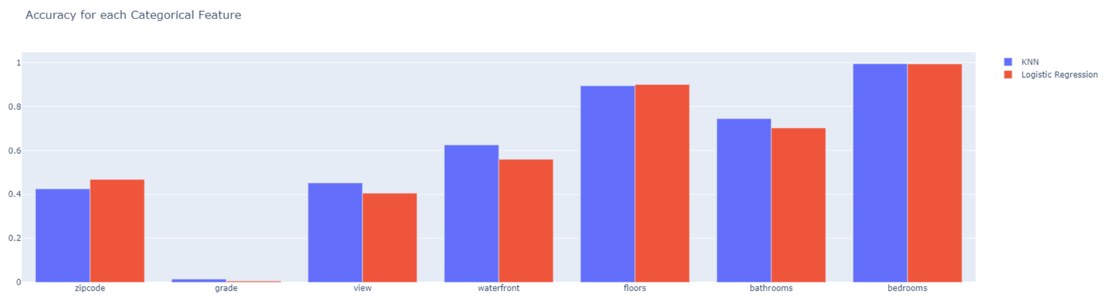


Figure 6: Accuracy for each Categorical Feature

We experimented on other orders to fill in the features. We sorted the features by their correlation with the price and then predicted them sequentially. The results are shown in Figure 7 and Figure 8.

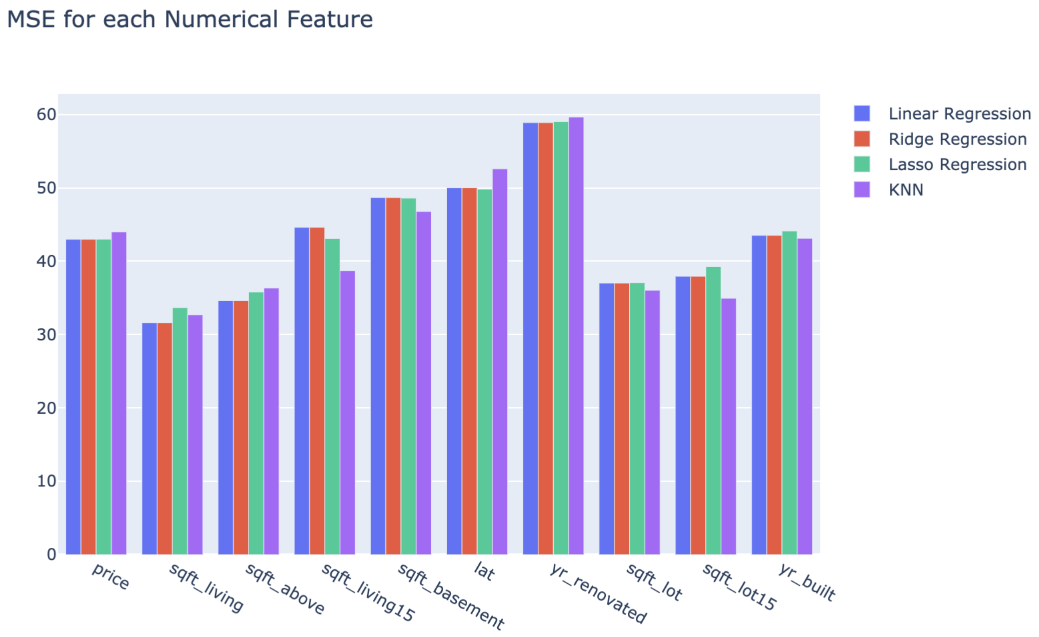


Figure 7: MSE for each Numerical Feature (Correlation Order)

For the regression results, the performance overall does not change much. While for the classification results, the performance improves in many cases. The order of features prediction can have an influence of this pipeline.

### 2.4.2 Denoising Autoencoder

First, we show the $MSE$ of the centered features for all reconstructed records in Figure 9:

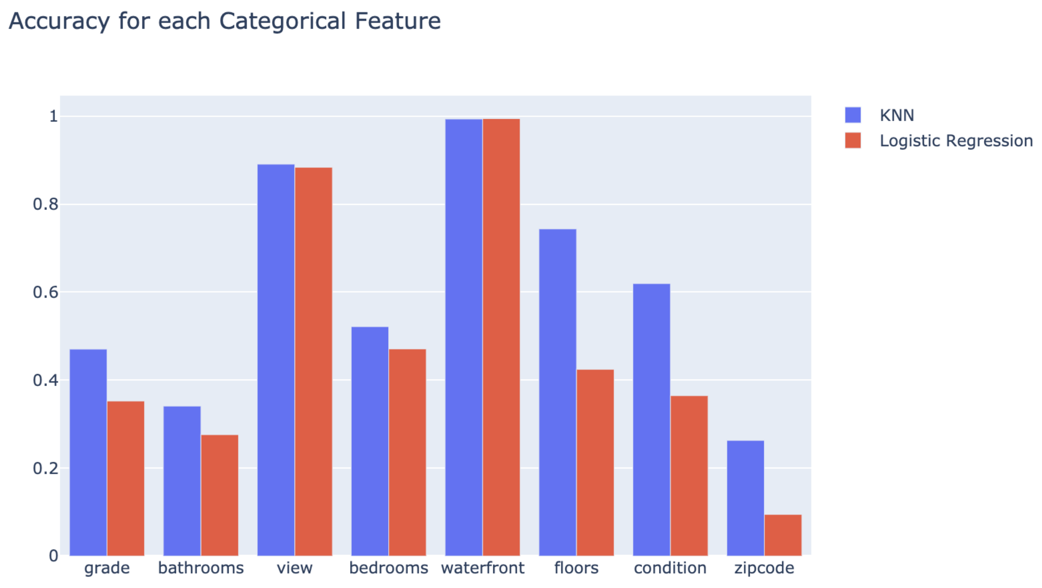


Figure 8: Accuracy for each Categorical Feature (Correlation Order)

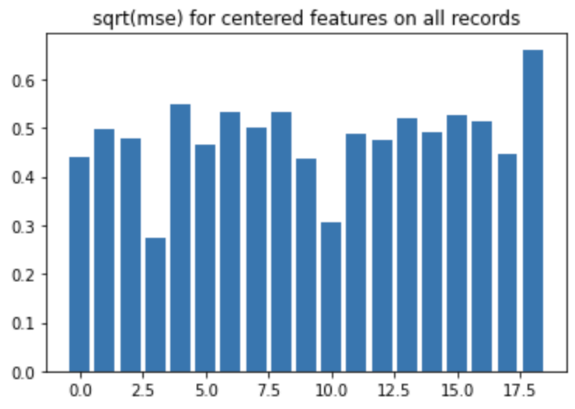


Figure 9: $\sqrt{MSE}$ of centered features on all records

In Figure 10, we show the $MSE$ of the centered features for missing values:

However, since the features have been normalized, it is hard for us to obtain a conclusion about how the model performs for each feature based on the $MSE$ plot above. So, it would be a good idea to convert the output back, remove the normalization and calculate the $MSE$. If we only use the naive $MSE$ on the converted data, because of the difference on our data scales, we are likely to get results like this, as shown in Figure 11:

To address the problem, we propose the relative mean square error, which take the scale of the data into consideration and make the $MSE$ on each feature comparable. Here are the results (Figure 12 and Figure 13):

From Figure 12 and Figure 13, we can see that the model performs relatively badly on feature 6, which is the number of floors. In my opinion, number of the floors is discrete value with small scale. So there might issues when the model tried to predict the floors as numerical data.

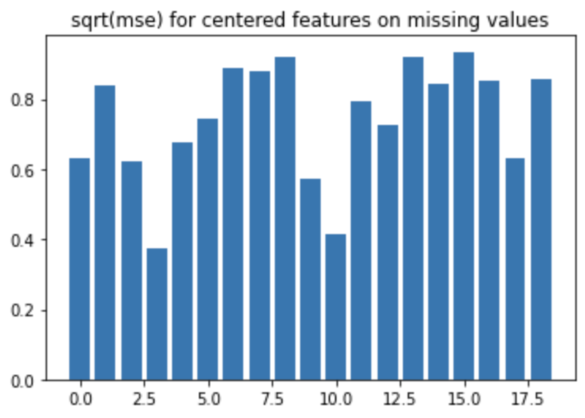


Figure 10: $\sqrt{MSE}$ of centered features on missing values

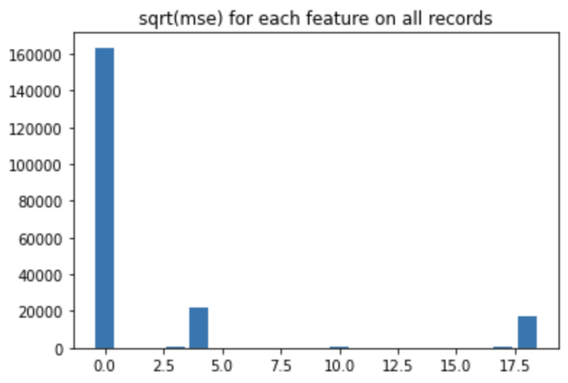


Figure 11: $\sqrt{MSE}$ of features on all records

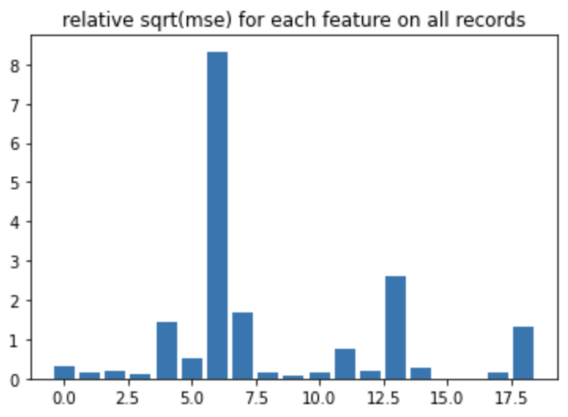


Figure 12: $\sqrt{relative\ MSE}$ of features on all records

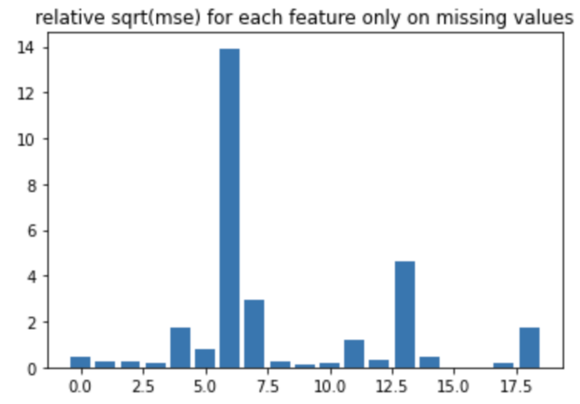


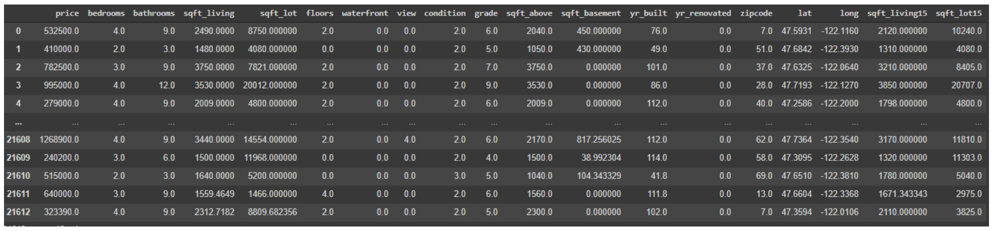
Figure 13: $\sqrt{relative\ MSE}$ of features on missing values

It is difficult for us to identify the significance of each predictor in our neural network. The original propose of this data set is to predict the house price. And from the result we can tell that the estimation on the missing price data is good by using other features in the data set.

## 2.5 Generate Data

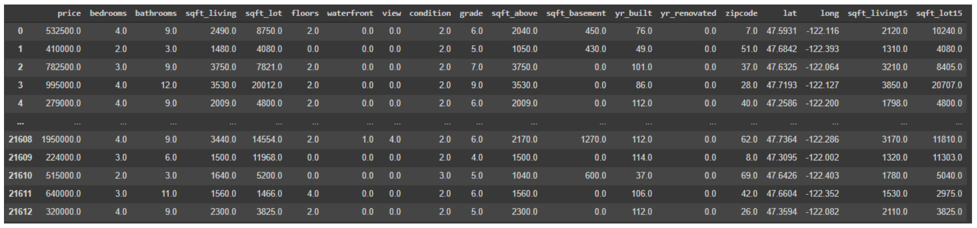
### 2.5.1 Models except Denoising Autoencoder

Since we used normalized features for the experiment, we denormalized the generated values so that we can compare them with the ground truth. The sample generated result is in shown in Figure 14. Compared with the ground truth (Figure 15), we can see the generated records look quite similar to the true ones.



| | price | bedrooms | bathrooms | sqft_living | sqft_lot | floors | waterfront | view | condition | grade | sqft_above | sqft_basement | yr_built | yr_renovated | zipcode | lat | long | sqft_living15 | sqft_lot15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 532500.0 | 4.0 | 9.0 | 2490.0000 | 8750.000000 | 2.0 | 0.0 | 0.0 | 2.0 | 6.0 | 2040.0 | 450.000000 | 76.0 | 0.0 | 7.0 | 47.5931 | -122.1160 | 2120.000000 | 10240.0 |
| 1 | 410000.0 | 2.0 | 3.0 | 1480.0000 | 4080.000000 | 0.0 | 0.0 | 0.0 | 2.0 | 5.0 | 1050.0 | 430.000000 | 49.0 | 0.0 | 51.0 | 47.6842 | -122.3930 | 1310.000000 | 4080.0 |
| 2 | 782500.0 | 3.0 | 9.0 | 3750.0000 | 7821.000000 | 2.0 | 0.0 | 0.0 | 2.0 | 7.0 | 3750.0 | 0.000000 | 101.0 | 0.0 | 37.0 | 47.6325 | -122.0640 | 3210.000000 | 8405.0 |
| 3 | 995000.0 | 4.0 | 12.0 | 3530.0000 | 20012.000000 | 2.0 | 0.0 | 0.0 | 2.0 | 9.0 | 3530.0 | 0.000000 | 86.0 | 0.0 | 28.0 | 47.7193 | -122.1270 | 3850.000000 | 20707.0 |
| 4 | 279000.0 | 4.0 | 9.0 | 2009.0000 | 4800.000000 | 2.0 | 0.0 | 0.0 | 2.0 | 6.0 | 2009.0 | 0.000000 | 112.0 | 0.0 | 40.0 | 47.2586 | -122.2000 | 1798.000000 | 4800.0 |
| ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... |
| 21608 | 1268900.0 | 4.0 | 9.0 | 3440.0000 | 14554.000000 | 2.0 | 0.0 | 4.0 | 2.0 | 6.0 | 2170.0 | 817.256025 | 112.0 | 0.0 | 62.0 | 47.7364 | -122.3540 | 3170.000000 | 11810.0 |
| 21609 | 240200.0 | 3.0 | 6.0 | 1500.0000 | 11968.000000 | 0.0 | 0.0 | 0.0 | 2.0 | 4.0 | 1500.0 | 38.992304 | 114.0 | 0.0 | 58.0 | 47.3095 | -122.2628 | 1320.000000 | 11303.0 |
| 21610 | 515000.0 | 2.0 | 3.0 | 1640.0000 | 5200.000000 | 0.0 | 0.0 | 0.0 | 3.0 | 5.0 | 1040.0 | 104.343329 | 41.8 | 0.0 | 69.0 | 47.6510 | -122.3810 | 1780.000000 | 5040.0 |
| 21611 | 640000.0 | 3.0 | 9.0 | 1559.4649 | 1466.000000 | 4.0 | 0.0 | 0.0 | 2.0 | 6.0 | 1560.0 | 0.000000 | 111.8 | 0.0 | 13.0 | 47.6604 | -122.3368 | 1671.343343 | 2975.0 |
| 21612 | 323390.0 | 4.0 | 9.0 | 2312.7182 | 8809.682356 | 2.0 | 0.0 | 0.0 | 2.0 | 5.0 | 2300.0 | 0.000000 | 102.0 | 0.0 | 7.0 | 47.3594 | -122.0106 | 2110.000000 | 3825.0 |

Figure 14: Interpolated Data



| | price | bedrooms | bathrooms | sqft_living | sqft_lot | floors | waterfront | view | condition | grade | sqft_above | sqft_basement | yr_built | yr_renovated | zipcode | lat | long | sqft_living15 | sqft_lot15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 532500.0 | 4.0 | 9.0 | 2490.0 | 8750.0 | 2.0 | 0.0 | 0.0 | 2.0 | 6.0 | 2040.0 | 450.0 | 76.0 | 0.0 | 7.0 | 47.5931 | -122.116 | 2120.0 | 10240.0 |
| 1 | 410000.0 | 2.0 | 3.0 | 1480.0 | 4080.0 | 0.0 | 0.0 | 0.0 | 2.0 | 5.0 | 1050.0 | 430.0 | 49.0 | 0.0 | 51.0 | 47.6842 | -122.393 | 1310.0 | 4080.0 |
| 2 | 782500.0 | 3.0 | 9.0 | 3750.0 | 7821.0 | 2.0 | 0.0 | 0.0 | 2.0 | 7.0 | 3750.0 | 0.0 | 101.0 | 0.0 | 37.0 | 47.6325 | -122.064 | 3210.0 | 8405.0 |
| 3 | 995000.0 | 4.0 | 12.0 | 3530.0 | 20012.0 | 2.0 | 0.0 | 0.0 | 2.0 | 9.0 | 3530.0 | 0.0 | 86.0 | 0.0 | 28.0 | 47.7193 | -122.127 | 3850.0 | 20707.0 |
| 4 | 279000.0 | 4.0 | 9.0 | 2009.0 | 4800.0 | 2.0 | 0.0 | 0.0 | 2.0 | 6.0 | 2009.0 | 0.0 | 112.0 | 0.0 | 40.0 | 47.2586 | -122.200 | 1798.0 | 4800.0 |
| ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... |
| 21608 | 1950000.0 | 4.0 | 9.0 | 3440.0 | 14554.0 | 2.0 | 1.0 | 4.0 | 2.0 | 6.0 | 2170.0 | 1270.0 | 112.0 | 0.0 | 62.0 | 47.7364 | -122.286 | 3170.0 | 11810.0 |
| 21609 | 224000.0 | 3.0 | 6.0 | 1500.0 | 11968.0 | 0.0 | 0.0 | 0.0 | 2.0 | 4.0 | 1500.0 | 0.0 | 114.0 | 0.0 | 8.0 | 47.3095 | -122.002 | 1320.0 | 11303.0 |
| 21610 | 515000.0 | 2.0 | 3.0 | 1640.0 | 5200.0 | 0.0 | 0.0 | 0.0 | 3.0 | 5.0 | 1040.0 | 600.0 | 37.0 | 0.0 | 69.0 | 47.6426 | -122.403 | 1780.0 | 5040.0 |
| 21611 | 640000.0 | 3.0 | 11.0 | 1560.0 | 1466.0 | 4.0 | 0.0 | 0.0 | 2.0 | 6.0 | 1560.0 | 0.0 | 106.0 | 0.0 | 42.0 | 47.6604 | -122.352 | 1530.0 | 2975.0 |
| 21612 | 320000.0 | 4.0 | 9.0 | 2300.0 | 3825.0 | 2.0 | 0.0 | 0.0 | 2.0 | 5.0 | 2300.0 | 0.0 | 112.0 | 0.0 | 26.0 | 47.3594 | -122.082 | 2110.0 | 3825.0 |

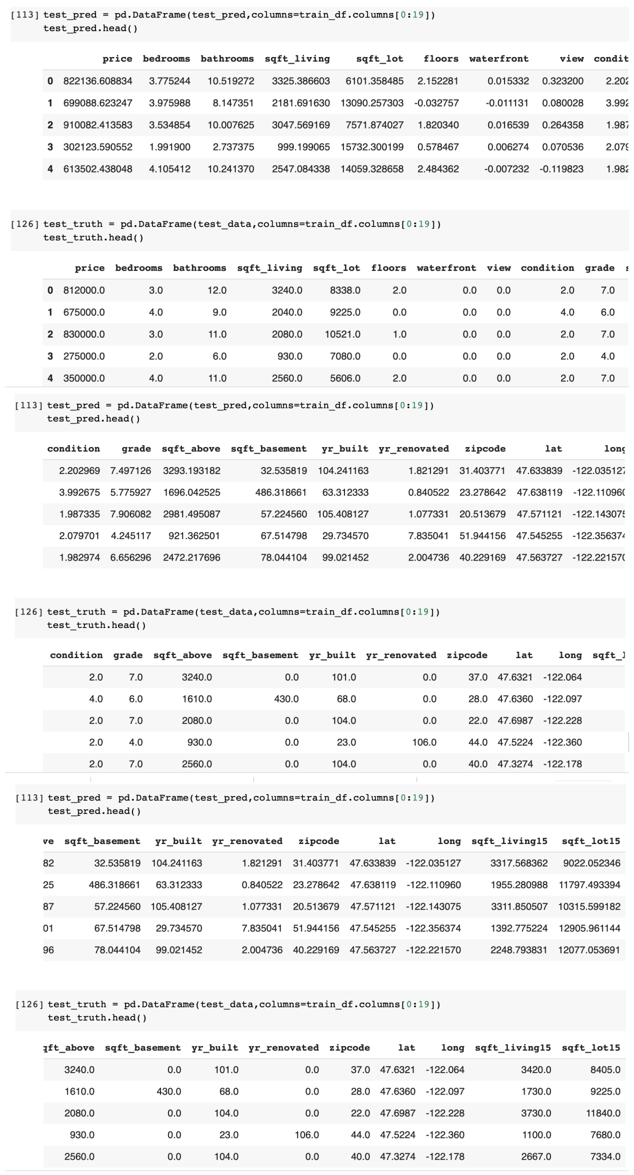
Figure 15: Original Data

### 2.5.2 Denoising Autoencoder

We can generate new data by entering records which have both realistic feature values and missing values to the model. For example, we can do the following steps:

1. Calculate the mean vector of all the records.
2. Add some noises to this vector. Or randomly drop some values from this vector.
3. Use this vector as the input for the auto encoder and generate a new record.

For the experiment, we show some example output when the input is part of the test data. We presented these examples so we can compare them with the ground truth to see if these reconstruction records look real.



```
[113] test_pred = pd.DataFrame(test_pred,columns=train_df.columns[0:19])
      test_pred.head()
```

| | price | bedrooms | bathrooms | sqft_living | sqft_lot | floors | waterfront | view | condit |
|---|---|---|---|---|---|---|---|---|---|
| **0** | 822136.608834 | 3.775244 | 10.519272 | 3325.386603 | 6101.358485 | 2.152281 | 0.015332 | 0.323200 | 2.202 |
| **1** | 699088.623247 | 3.975988 | 8.147351 | 2181.691630 | 13090.257303 | -0.032757 | -0.011131 | 0.080028 | 3.992 |
| **2** | 910082.413583 | 3.534854 | 10.007625 | 3047.569169 | 7571.874027 | 1.820340 | 0.016539 | 0.264358 | 1.987 |
| **3** | 302123.590552 | 1.991900 | 2.737375 | 999.199065 | 15732.300199 | 0.578467 | 0.006274 | 0.070536 | 2.079 |
| **4** | 613502.438048 | 4.105412 | 10.241370 | 2547.084338 | 14059.328658 | 2.484362 | -0.007232 | -0.119823 | 1.982 |

```
[126] test_truth = pd.DataFrame(test_data,columns=train_df.columns[0:19])
      test_truth.head()
```

| | price | bedrooms | bathrooms | sqft_living | sqft_lot | floors | waterfront | view | condition | grade | s |
|---|---|---|---|---|---|---|---|---|---|---|---|
| **0** | 812000.0 | 3.0 | 12.0 | 3240.0 | 8338.0 | 2.0 | 0.0 | 0.0 | 2.0 | 7.0 | |
| **1** | 675000.0 | 4.0 | 9.0 | 2040.0 | 9225.0 | 0.0 | 0.0 | 0.0 | 4.0 | 6.0 | |
| **2** | 830000.0 | 3.0 | 11.0 | 2080.0 | 10521.0 | 1.0 | 0.0 | 0.0 | 2.0 | 7.0 | |
| **3** | 275000.0 | 2.0 | 6.0 | 930.0 | 7080.0 | 0.0 | 0.0 | 0.0 | 2.0 | 4.0 | |
| **4** | 350000.0 | 4.0 | 11.0 | 2560.0 | 5606.0 | 2.0 | 0.0 | 0.0 | 2.0 | 7.0 | |

```
[113] test_pred = pd.DataFrame(test_pred,columns=train_df.columns[0:19])
      test_pred.head()
```

| condition | grade | sqft_above | sqft_basement | yr_built | yr_renovated | zipcode | lat | long |
|---|---|---|---|---|---|---|---|---|
| 2.202969 | 7.497126 | 3293.193182 | 32.535819 | 104.241163 | 1.821291 | 31.403771 | 47.633839 | -122.035127 |
| 3.992675 | 5.775927 | 1696.042525 | 486.318661 | 63.312333 | 0.840522 | 23.278642 | 47.638119 | -122.110960 |
| 1.987335 | 7.906082 | 2981.495087 | 57.224560 | 105.408127 | 1.077331 | 20.513679 | 47.571121 | -122.143075 |
| 2.079701 | 4.245117 | 921.362501 | 67.514798 | 29.734570 | 7.835041 | 51.944156 | 47.545255 | -122.356374 |
| 1.982974 | 6.656296 | 2472.217696 | 78.044104 | 99.021452 | 2.004736 | 40.229169 | 47.563727 | -122.221570 |

```
[126] test_truth = pd.DataFrame(test_data,columns=train_df.columns[0:19])
      test_truth.head()
```

| condition | grade | sqft_above | sqft_basement | yr_built | yr_renovated | zipcode | lat | long | sqft_l |
|---|---|---|---|---|---|---|---|---|---|
| 2.0 | 7.0 | 3240.0 | 0.0 | 101.0 | 0.0 | 37.0 | 47.6321 | -122.064 | |
| 4.0 | 6.0 | 1610.0 | 430.0 | 68.0 | 0.0 | 28.0 | 47.6360 | -122.097 | |
| 2.0 | 7.0 | 2080.0 | 0.0 | 104.0 | 0.0 | 22.0 | 47.6987 | -122.228 | |
| 2.0 | 4.0 | 930.0 | 0.0 | 23.0 | 106.0 | 44.0 | 47.5224 | -122.360 | |
| 2.0 | 7.0 | 2560.0 | 0.0 | 104.0 | 0.0 | 40.0 | 47.3274 | -122.178 | |

```
[113] test_pred = pd.DataFrame(test_pred,columns=train_df.columns[0:19])
      test_pred.head()
```

| ve | sqft_basement | yr_built | yr_renovated | zipcode | lat | long | sqft_living15 | sqft_lot15 |
|---|---|---|---|---|---|---|---|---|
| 82 | 32.535819 | 104.241163 | 1.821291 | 31.403771 | 47.633839 | -122.035127 | 3317.568362 | 9022.052346 |
| 25 | 486.318661 | 63.312333 | 0.840522 | 23.278642 | 47.638119 | -122.110960 | 1955.280988 | 11797.493394 |
| 87 | 57.224560 | 105.408127 | 1.077331 | 20.513679 | 47.571121 | -122.143075 | 3311.850507 | 10315.599182 |
| 01 | 67.514798 | 29.734570 | 7.835041 | 51.944156 | 47.545255 | -122.356374 | 1392.775224 | 12905.961144 |
| 96 | 78.044104 | 99.021452 | 2.004736 | 40.229169 | 47.563727 | -122.221570 | 2248.793831 | 12077.053691 |

```
[126] test_truth = pd.DataFrame(test_data,columns=train_df.columns[0:19])
      test_truth.head()
```

| qft_above | sqft_basement | yr_built | yr_renovated | zipcode | lat | long | sqft_living15 | sqft_lot15 |
|---|---|---|---|---|---|---|---|---|
| 3240.0 | 0.0 | 101.0 | 0.0 | 37.0 | 47.6321 | -122.064 | 3420.0 | 8405.0 |
| 1610.0 | 430.0 | 68.0 | 0.0 | 28.0 | 47.6360 | -122.097 | 1730.0 | 9225.0 |
| 2080.0 | 0.0 | 104.0 | 0.0 | 22.0 | 47.6987 | -122.228 | 3730.0 | 11840.0 |
| 930.0 | 0.0 | 23.0 | 106.0 | 44.0 | 47.5224 | -122.360 | 1100.0 | 7680.0 |
| 2560.0 | 0.0 | 104.0 | 0.0 | 40.0 | 47.3274 | -122.178 | 2667.0 | 7334.0 |

Figure 16: Compare generated records and true records

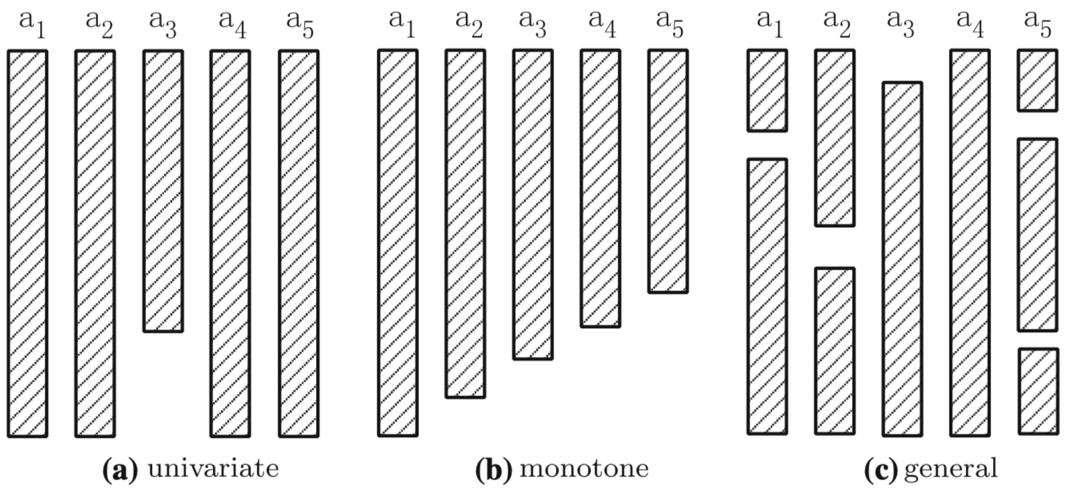
From the results, we can tell the generated records are similar with the true records. And we can make the record more realistic by converting some numerical values to categorical values, such as keeping the integer part of the bedroom number estimation or grade value.

The generated record looks good. And this means the auto encoder captures the underlying distribution of the data well and is able to represent the record in the latent space.

## 2.6 Conclusion

### 2.6.1 Model Comparison

We can see that denoising auto encoder shows some advantages over the first model pipeline. One advantage of the auto encoder is that denoising auto encoder takes all the present features in the record into consideration and perform the prediction. While in the first pipeline, we predict the missing values sequentially. At the beginning, the information for models in pipeline 1 is very limited, thus will lead to a bad performance.



$a_1$ $a_2$ $a_3$ $a_4$ $a_5$
$a_1$ $a_2$ $a_3$ $a_4$ $a_5$
$a_1$ $a_2$ $a_3$ $a_4$ $a_5$
(a) univariate
(b) monotone
(c) general

**Fig. 5** Common missing patterns

Figure 17: Common Missing Value Pattern (fig.5 in [1])

However, in some cases, pipeline 1 will have the potential to perform better. For instance, when the missing values obey the monotone pattern, the best solution to perform data interpolation will be predicting missing values sequentially, from the column with least missing values to column with most missing values. In this case, pipeline 1 will be potentially successful.

### 2.6.2 Improvement

For both methods, we have some ideas to improve it:

For the model pipeline 1, we can choose the order of predicting features wisely. Moreover, we could predict the missing values multiple times using different orders. Then choose the average values as the final prediction.

For the denoising autoencoder, we can customize our loss function to our mixed type data. For example, for categorical data, we can use one hot embedding instead of label embedding. And then use cross entropy loss for the categorical data. For other numerical data, we still use $MSE$ loss. We assign suitable weights on these two parts and add them together as our final loss function. In this way we can address the problem of mixed data properly.

Another way to improve the denoising autoencoder is to improve the structure of the network. We can add more functions such as drop out method and L1/L2 regularization. Or just add more layers to the network.

## 3 Reference

[1] MIDIA: exploring denoising autoencoders for missing data imputation, Qian Ma, Wang-Chien Lee, Tao-Yang Fu, Yu Gu, Ge Yu, Data Mining and Knowledge Discovery (2020) 34:1859–1897.

[2] Neural Network from scratch in Python, Omar Aflak, towards data science blog.