Yuqi Cheng
yc2233

Homework 1

CS 5785
September 16, 2021

Programming Exercises

Part I The Housing Price

(1) Here we get 79 features in total, and we can easily find there are two kinds of features, categorical and continuous. We give 3 examples of each type as follows,

Continuous feature examples:

- LotFrontage: Linear feet of street connected to property
- LotArea: Lot size in square feet
- GarageArea: Size of garage in square feet

Categorical feature examples:

- MSZoning: Identifies the general zoning classification of the sale.
- Street: Type of road access to property.
- Alley: Type of alley access to property.

We also create plots for two example features we choose.

```
In [6]: #choose LotFrontage as the example of continuous features
        LF = train_data['LotFrontage']
        plt.hist(LF, bins=20, width = 10)
        plt.title('Distribution of LotFrontage')
        plt.ylabel("Linear feet", fontsize=12)
        plt.ylabel("Frequency", fontsize=12)
Out[6]: Text(0, 0.5, 'Frequency')
```

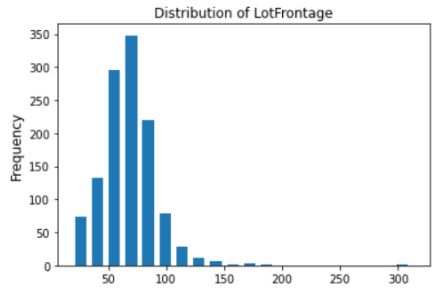

Figure 1: Distribution of LotFrontage

```
In [7]: #choose LotFrontage as the example of categorical features
        St = train_data['Street']
        tmp = {}
        for i in np.unique(St):
            tmp[i] = np.sum(St == i)

        names = list(tmp.keys())
        values = list(tmp.values())

        plt.bar(names, values, width = 0.5, color = 'orange')
        plt.title('Distribution of Street Type')
        plt.ylabel("Street Type", fontsize=12)
        plt.ylabel("Frequency", fontsize=12)
Out[7]: Text(0, 0.5, 'Frequency')
```

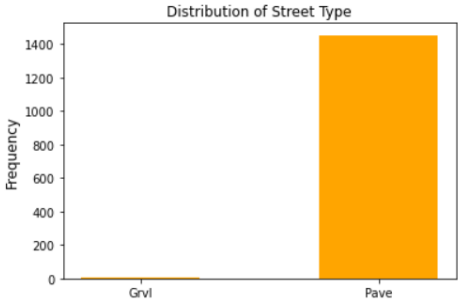

Figure 2: Distribution of Street

(2) We then preprocess our dataset, to give a high quality dataset to our model, we do the following steps:

1. Filter and fill missing values (NaN)
2. Do One-hot encoding for all categorical features
3. Normalize all continuous data
4. Apply PCA on all features to do dimension reduction and eliminate multicollinearity

(2.1) Filter and fill missing values

Interestingly, we find not all NaN represent missing value, for example, NA in Alley means no alley access. So we use **None** replace NaN.

```
In [12]: for i in na_with_ft:
             total[i].fillna('None', inplace=True)

In [13]: all_na_remain = (total.isnull().sum() / len(total)) * 100
         all_na_remain_num = len(all_na_remain[all_na_remain != 0]) - 1
         print("We get",all_na_remain_num,"features that have NaN value in total.")

         We get 20 features that have NaN value in total.
```

Figure 3: Replace meaningful NaN

After processing the meaningful NaN values, we get 20 (except SalePrice) features still have missing values. Among these 20 features,

- MSZoning, Utilities, Exterior1st, Exterior2nd, MasVnrType, Electrical, KitchenQual, Functional, and SaleType are categorical data with object Dtype.
- BsmtFullBath and BsmtHalfBath are categorical data with float Dtype.
- For categorical data, we use the most frequent occurred category to fill the missing value.

For categorical data, we use the most frequent occurred category to fill the missing value.

```
In [15]: #process categorical data
         ct_data = ['MSZoning', 'Utilities', 'Exterior1st', 'Exterior2nd', 'MasVnrType',
                    'Electrical', 'KitchenQual', 'Functional', 'SaleType','BsmtFullBath','BsmtHalfBath']

         for i in ct_data:
             imp_mode = total[i].mode()[0]
             total[i].fillna(imp_mode, inplace=True)
```

Figure 4: Categorical missing value filling

For continuous data, we use mean value to fill the missing value.

```
In [16]: cn_data = ['LotFrontage','MasVnrArea','BsmtFinSF1','BsmtFinSF2',
                    'BsmtUnfSF','TotalBsmtSF','GarageYrBlt','GarageCars','GarageArea']

In [17]: for i in cn_data:
             imp_mean = total[i].mean()
             total[i].fillna(imp_mean, inplace=True)

In [18]: all_na_remain = (total.isnull().sum() / len(total)) * 100
         all_na_remain_num = len(all_na_remain[all_na_remain != 0]) - 1
         print("We get",all_na_remain_num,"features that have NaN value in total.")

         We get 0 features that have NaN value in total.
```

Figure 5: Continuous missing value filling

(2.2) One-hot encoding for all categorical features.

We first turn MSSubClass to string data type for this feature is a categorical feature though it has numerical items.

All the categorical features need to be converted to one-hot encoding. After careful research, we found that all these data are not prioritized, and if we do not use one-hot encoding, these data will be marked as 1, 2, 3, so that there is a sequential relationship, which will affect the model Training results. Therefore, we need to use one-hot encoding to process these data.

Take Street as an example again:

```
In [7]: #choose LotFrontage as the example of categorical features
        St = train_data['Street']
        tmp = {}
        for i in np.unique(St):
            tmp[i] = np.sum(St == i)

        names = list(tmp.keys())
        values = list(tmp.values())

        plt.bar(names, values, width = 0.5, color = 'orange')
        plt.title('Distribution of Street Type')
        plt.ylabel("Street Type", fontsize=12)
        plt.ylabel("Frequency", fontsize=12)

Out[7]: Text(0, 0.5, 'Frequency')
```

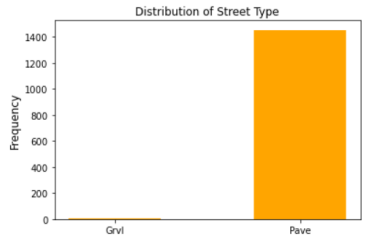

Figure 6: Before OHE

```
In [23]: plt.hist(st_ohe)

Out[23]: (array([[2907.,    0.,    0.,    0.,    0.,    0.,    0.,    0.,
                    12.],
                 [  12.,    0.,    0.,    0.,    0.,    0.,    0.,    0.,
                  2907.]]),
          array([0. , 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1. ]),
          <a list of 2 BarContainer objects>)
```

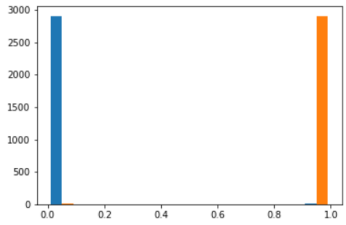

Figure 7: After OHE

We then convert all categorical data into OHE format.

(2.3) Normalize all continuous data

Meanwhile, we find the continuous data are with different scales. For example, LotFrontage is scaled in feet while YearBuilt is a time series data. So we use Z-score standardization.

```
In [26]: target = total_ohe['SalePrice']

In [27]: total_ohe = total_ohe.drop(labels='SalePrice', axis=1)

In [28]: for i in total_ohe.columns:
             if total_ohe[i].dtype!='uint8':
                 total_ohe[i] = scale(total_ohe[i])
```

Figure 8: Normalization

(2.4) Apply PCA (Principle Component Analysis) on all features

We do not choose a set of specific features to fit our model because we are lack of professional knowledge of Housing price. Thus we choose to use all features, all of which seems useful and reasonable. So the problems we meet are multicollinearity and a huge dimension.

For the dimension here is too high, we use PCA to do dimension reduction. And PCA can also eliminate multicollinearity.

We first explore the cumulative explained variance ratio of the result of PCA.

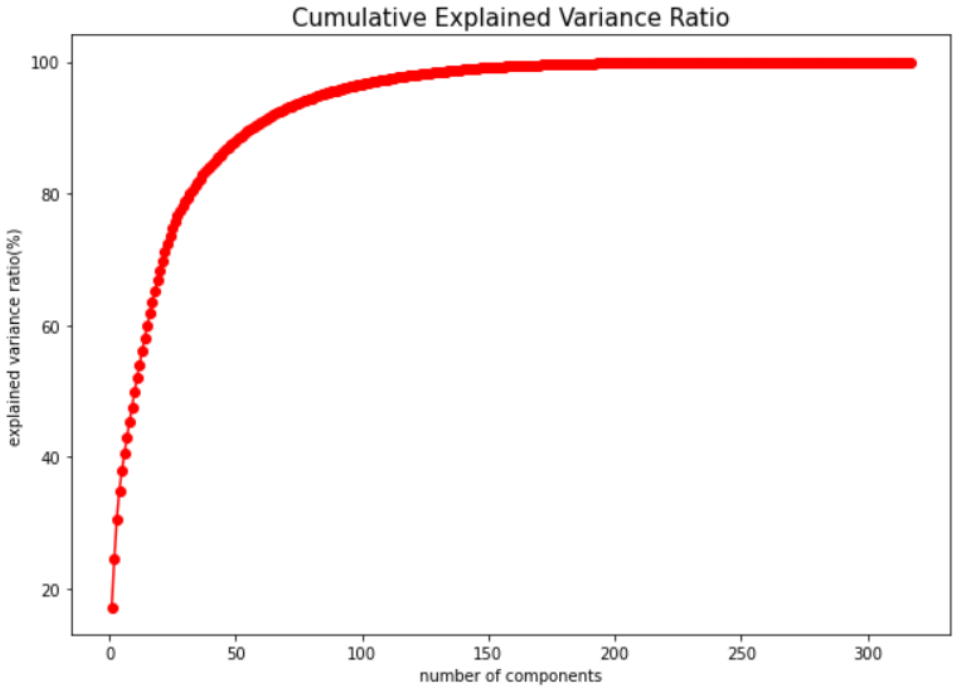

Figure 9: Explained variance ratio

We choose to keep 90% variance after PCA.

```
In [30]: set_var = 0.9 #keep 90% variance
         total_ohe_pca = PCA(n_components=set_var).fit_transform(total_ohe)

         print("original shape: ", total_ohe.shape)
         print("transformed shape: ", total_ohe_pca.shape)

         original shape:  (2919, 317)
         transformed shape:  (2919, 57)
```

Figure 10: PCA

After PCA, we get 57 dimensions in our processed dataset.

(3) Using ordinary least squares (OLS) to predict house prices

We implement Ordinary least squares, MSE and R Square without any packages

```
In [35]: class LR(object):

             def __init__(self):
                 self.coef = None
                 self.interception = None
                 self.theta = None

             def fit(self, X_train, y_train):
                 assert X_train.shape[0] == y_train.shape[0] #the size of X_tra

                 X = np.hstack([np.ones((len(X_train), 1)), X_train])
                 self.theta = np.linalg.inv(X.T.dot(X)).dot(X.T).dot(y_train)

                 self.interception = self.theta[0]
                 self.coef = self.theta[1:]

                 return self

             def predict(self, X_predict):
                 assert X_predict.shape[1] == len(self.coef)

                 X = np.hstack([np.ones((len(X_predict), 1)), X_predict])
                 return X.dot(self.theta)

         def MSE(pred, test):
             assert pred.shape[0] == test.shape[0]
             return np.sum((pred-test)**2) / len(pred)

         def R2(pred, test):
             mse = MSE(pred, test)
             return (1-mse/np.var(test))
```

Figure 11: OLS

Meanwhile, we use sklearn to split train set and test set. The training result is shown as follows,

```
In  [40]: y_predict = reg.predict(X_test)

In  [41]: MSE(y_predict, y_test)
Out[41]: 637197675.9930685

In  [42]: R2(y_predict, y_test)
Out[42]: 0.8914542084850577
```

Figure 12: Model performance

where $R^2 = 0.89$

(4) Submit to Kaggle

Finally, we submit our prediction to Kaggle, the score we get is **0.17**

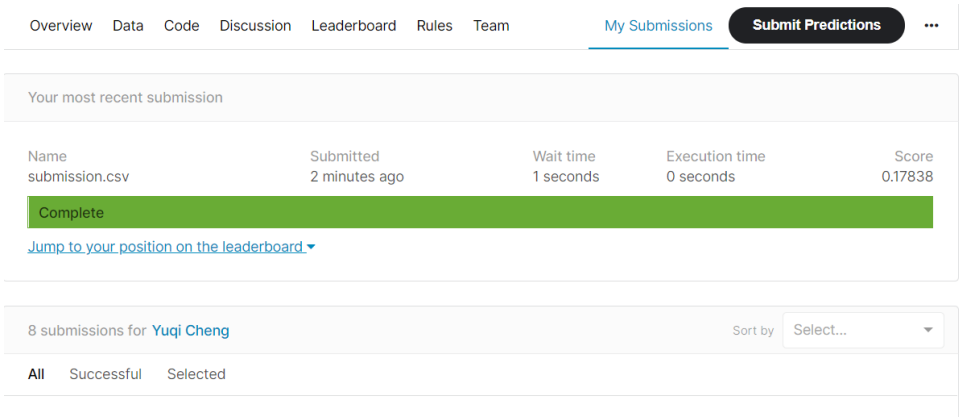

Figure 13: Kaggle score

Part II The Titanic Disaster

In this part, as same as the previous part, we first download the datasets and preprocess them and apply our model. What's different is this part presents a classification problem while the Part I is a regression problem.

In this part, we use **Logistic Regression** to fit the data and use **Accuarcy** to evaluate the model performance rather than MSE.

The outline of our solution is:

1. Filter and fill missing values (NaN)
2. Feature selection
3. Do One-hot encoding for all categorical features
4. Normalize all continuous data
5. Apply PCA on all remaining features
6. Apply Logistic Regression to fit the data

(1) Filter and fill missing values (NaN)

Once we obtain the dataset, after loading them, we first try to detect if there were any features that have missing value.

```
In [21]: #calculate how many features have missing value
         all_na = (total.isnull().sum() / len(total)) * 100
         all_na_num = len(all_na[all_na != 0])
         #all_na[all_na!=0]
         print("We get",all_na_num,"features that have NaN value in total.")

         We get 4 features that have NaN value in total.
```

Figure 14: Missing value detection

Here, we have 4 features have NaN. Different with what we did in part 1, the dataset in this part dose not have meaningful NaN value. So we can fill and filter these features directly.

- we drop the entire 'cabin' column, since there's no way to replace the null based on the given information and the missing values are too many to fill in and Cabin number is the passenger's personal choice.

```
In [22]: total = total.drop(labels='Cabin', axis=1)
         total = total.drop(labels='Name', axis=1)

In [23]: total = total.drop(labels='Ticket',axis=1)

In [24]: #calculate how many features have missing value
         all_na_remain = (total.isnull().sum() / len(total)) * 100
         all_na_num_remain = len(all_na_remain[all_na_remain != 0])
         #all_na[all_na!=0]
         print("We get",all_na_num_remain,"features that have NaN value in total.")

         We get 3 features that have NaN value in total.
```

Figure 15: Remove Cabin feature

Now we have 3 features with missing values remained. **Age**,**Fare** and **Embarked**.

In all these features with missing value:

- Age and Fare are continuous data
- Embarked is categorical data

For categorical data, we use the most frequent occurred category to fill the missing value. And For continuous data, we use mean value to fill the missing value.

```
In [28]: total.head()
```

Out[28]:

PassengerId	Pclass	Sex	Age	SibSp	Parch	Fare	Embarked
1	3	male	22.0	1	0	7.2500	S
2	1	female	38.0	1	0	71.2833	C
3	3	female	26.0	0	0	7.9250	S
4	1	female	35.0	1	0	53.1000	S
5	3	male	35.0	0	0	8.0500	S

```
In [29]: #calculate how many features have missing value
         all_na_remain = (total.isnull().sum() / len(total)) * 100
         all_na_num_remain = len(all_na_remain[all_na_remain != 0])
         #all_na[all_na!=0]
         print("We get",all_na_num_remain,"features that have NaN value in total.")

         We get 0 features that have NaN value in total.
```

Figure 16: All missing values are cleared

(2) Feature selection

We also find that there are some features is not relative with the survival rate, like a good name and lucky ticket number can not prevent one from death in the disaster. Here, we also drop the entire 'name' and 'Ticket' column, which are all unique identification values, no relationship with the prediction.

We use the **Pclass, Sex, Age, SibSp, Parch, Fare and Embarked** features to do prediction.

```
In [22]: total = total.drop(labels='Cabin', axis=1)
         total = total.drop(labels='Name', axis=1)

In [23]: total = total.drop(labels='Ticket',axis=1)
```

Figure 17: Feature selection

(3) One-hot encoding and normalization

Same with what we did in the previous part, we convert all categorical data into OHE format, and Z-score normalization for every continuous features.

```
In [32]: total_ohe = pd.get_dummies(total)

         Z-score normalization for every continuous features

In [33]: for i in total_ohe.columns:
             if total_ohe[i].dtype!='uint8':
                 total_ohe[i] = scale(total_ohe[i])
```

Figure 18: Feature selection

(4) PCA

We also apply PCA to the expanded feature space.

```
In [34]: set_var = 0.9 #keep 90% variance
         total_ohe_pca = PCA(n_components=set_var).fit_transform(total_ohe)

In [35]: total_ohe_pca.shape

Out[35]: (1309, 8)
```

Figure 19: PCA

(5) Logistic Regression

Here, we use sklearn package directly to apply logistic regression, and we also use sklearn to split train set and test set. The training result is shown as follows,

```
In [38]: from sklearn.linear_model import LogisticRegression
         from sklearn.metrics import accuracy_score
         from sklearn.model_selection import train_test_split

         Logistic regression

In [39]: X_train, X_test, y_train, y_test = train_test_split(train_data_filtered, train_label_filtered, test_size=0.2, random_state=11)

         #logistic regression
         Logit = LogisticRegression()
         Logit.fit(X_train, y_train)
         y_predict = Logit.predict(X_test)

         accuracy_score(y_predict, y_test)

Out[39]: 0.8659217877094972
```

Figure 20: Logistic Regression

Accuracy here is 0.87.

(6) Submit to Kaggle

Finally, we submit our prediction to Kaggle, the score we get is **0.77**

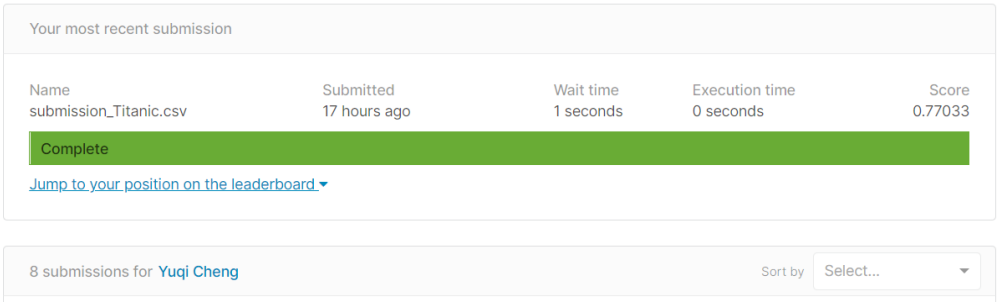

Figure 21: Kaggle score

Written Exercises

1. Connections between the KL divergence and maximum likelihood learning

We first explore the maxium likelihood, according to the empirical distribution, we can find that $\hat{p}(x, y)$ is a continuous distribution (uniform distribution). So we have,

$$\begin{aligned}
\arg\max_\theta \mathbb{E}_{\hat{p}(x,y)}[logp_\theta(y|x)] &= \arg\max_\theta \int_{x,y} \hat{p}(x,y) logp_\theta(y|x) dxdy \\
&= \arg\max_\theta \int_{x,y} \hat{p}(y|x)\hat{p}(x) logp_\theta(y|x) dxdy \\
&= \arg\max_\theta \int_x \hat{p}(x)[\int_y \hat{p}(y|x) logp_\theta(y|x) dy]dx \\
&= \arg\max_\theta E_{x\sim\hat{p}(x)}[\int_y \hat{p}(y|x) logp_\theta(y|x) dy] \\
&= \arg\min_\theta E_{x\sim\hat{p}(x)}[\int_y -\hat{p}(y|x) logp_\theta(y|x) dy]
\end{aligned} \tag{1}$$

Here we get the conditional cross entropy. And if we add a no-theta item, we will have

$$\arg\min_\theta E_{x\sim\hat{p}(x)}[\int_y -\hat{p}(y|x) logp_\theta(y|x) dy + \int_y \hat{p}(y|x) logp(y|x) dy] \tag{2}$$

where we have equation (1) = equation (2), then we try to simplify equation (2), we have

$$\begin{aligned}(2) &= \arg\min_{\theta} E_{x\sim\hat{p}(x)}[\int_y \hat{p}(y|x) log \frac{\hat{p}(y|x)}{p_\theta(y|x)} dy] \\ &= \arg\min_{\theta} E_{x\sim\hat{p}(x)}[KL(\hat{p}(y|x)||p_\theta(y|x))]\end{aligned} \tag{3}$$

Thus we have (1) equals (3), which is

$$\arg\max_{\theta} \mathbb{E}_{\hat{p}(x,y)}[log p_\theta(y|x)] = \arg\min_{\theta} E_{\hat{p}(x)}[KL(\hat{p}(y|x)||p_\theta(y|x))]$$

2. Gradient and log-likelihood for logistic regression

(a) In this section, we try to prove $\frac{d\sigma(a)}{da} = \sigma(a)(1-\sigma(a))$. We have been given $\sigma(a) = \frac{1}{1+e^{-a}}$. We have,

$$\frac{d\sigma(a)}{da} = \frac{d(\frac{1}{1+e^{-a}})}{da} = \frac{e^{-x}}{(1+e^{-x})^2}$$

Meanwhile, we have

$$\sigma(a)(1-\sigma(a) = \frac{1}{1+e^{-a}} \frac{(1+e^{-x})-1}{1+e^{-a}} = \frac{e^{-x}}{(1+e^{-x})^2}$$

Thus, we have

$$\frac{d\sigma(a)}{da} = \sigma(a)(1-\sigma(a))$$

(b) We then use the equation we proved in 2(a) to further prove $\nabla\ell(\theta) = [y - \sigma(\theta^T\mathbf{x})]\mathbf{x}$, where

$$\ell(\theta) = y log\sigma(\theta^T\mathbf{x}) + (1-y) log(1-\sigma(\theta^T\mathbf{x}))$$

We look into $\nabla\ell(\theta)$ and find this can be decomposed as,

$$\nabla\ell(\theta) = \frac{\partial\ell}{\partial p}\frac{\partial p}{\partial s}\frac{\partial s}{\partial\theta}$$

where, $p = \sigma(\theta^T\mathbf{x})$ and $s = \theta^T\mathbf{x}$. Thus, $\ell(\theta) = ylog(p) + (1-y)log(1-p)$ and $\sigma(\theta^T\mathbf{x}) = \sigma(s)$ Then we have,

$$\frac{\partial\ell}{\partial p} = \frac{y}{p} - \frac{1-y}{1-p}$$

$$\frac{\partial p}{\partial s} = \frac{\partial \sigma(s)}{\partial s} = \sigma(s)(1 - \sigma(s)$$

$$\frac{\partial s}{\partial \theta} = \mathbf{x}$$

Thus,

$$\begin{aligned}
\nabla \ell(\theta) &= (\frac{y}{p} - \frac{1-y}{1-p})\sigma(s)(1 - \sigma(s)\mathbf{x} \\
&= (\frac{y}{\sigma(s)} - \frac{1-y}{1-\sigma(s)})\sigma(s)(1 - \sigma(s)\mathbf{x} \\
&= [y(1 - \sigma(s)) - (1-y)\sigma(s)]\mathbf{x} \\
&= [y - \sigma(s)]\mathbf{x} \\
&= [y - \sigma(\theta^T \mathbf{x})]\mathbf{x}
\end{aligned}$$

Finally, we get $\nabla \ell(\theta) = [y - \sigma(\theta^T \mathbf{x})]\mathbf{x}$.

3. Linear regression and OLS

(a). We use MATLAB to plot the points (Fig 1)

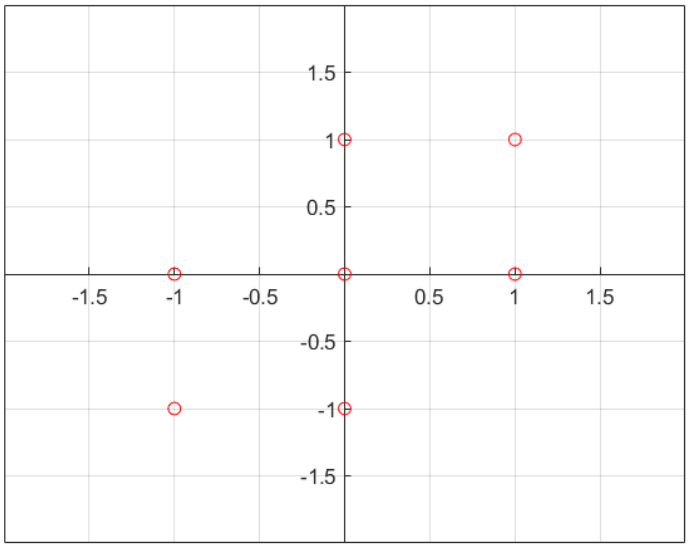

Figure 22: Scatter plot

Based on Fig 1, without any calculation, we think the **slope = 1** and **intercept = 0**.

(b). Through math calculation, our slope is 0.5, and intercept is 0 using the MSE.

According to MSE, we have

$$J(a,b) = \frac{1}{n}[y_i - (ax_i + b)^2]$$

$$\frac{\partial J(a,b)}{\partial b} = \frac{1}{n}\sum_{i=1}^{n} 2[y_i - (ax_i + b)](-1) = 0 \tag{4}$$

$$\frac{\partial J(a,b)}{\partial a} = \frac{1}{n}\sum_{i=1}^{n} 2[y_i - (ax_i + b)](-x_i) = 0 \tag{5}$$

According to Equation (4) and (5), we have

$$a = \frac{\sum_{i=1}^{n}(x_i - \overline{x})(y_i - \overline{y})}{\sum_{i=1}^{n}(x_i - \overline{x})^2}$$

$$b = \overline{y} - a\overline{x}$$

Thus we have

$$\overline{x} = \frac{-1 - 1 + 0 + 0 + 0 + 1 + 1}{7} = 0$$

$$\overline{y} = \frac{-1 + 0 - 1 + 0 + 1 + 0 + 1}{7} = 0$$

$$\overline{xy} = \frac{(-1 \times -1) + (-1 \times 0) + (0 \times -1) + (0 \times 0) + (0 \times 1) + (1 \times 0) + (1 \times 1)}{7} = \frac{2}{7}$$

$$\overline{x^2} = \frac{(-1 \times -1) + (-1 \times -1) + 0 + 0 + 0 + (1 \times 1) + (1 \times 1)}{7} = \frac{4}{7}$$

$$slope = \frac{0 - \frac{2}{7}}{0 - \frac{4}{7}} = \frac{1}{2}$$

$$intercept = 0 - 0 \times 0.5 = 0$$

In section (a), we think that the line $y = x$ is the best fitting line, since the distribution of the data points round the origin and seems like $y = x$ fits a good line across (-1,-1),(0,0) and (1,1).

However, we should also notice that the definition of the MSE is to compute the squared error and thus enlarge the error. Consequently, though y = 0.5x passes less points than line y = x, it's total error would be less using squared calculation.

(c) To use MAE as the loss function, we have $\frac{1}{n}\sum_{i=1}^{n}|y_i - \hat{y}_i|$ in which $\hat{y}_i$ is the predicted value and y_i is the true value.

In our dataset, we only have $x = \{-1, 0, 1\}$, $y = \{-1, 0, 1\}$. It is okay for us to calculate with specific examples. To illustrate,

- we can start with the line $y = 0$ which passes through three points $(-1, 0), (0, 0), (1, 0)$, in this case, MAE is $|-1-0| + |1-0| + |-1-0| + |-1-0| = 4$.
- And then, we take $y = x$ as another example. This line with $slope = 1$ also passes through three points $(-1, -1), (0, 0), (1, 1)$ and the MAE is $|0-(-1)| + |1-0| + |-1-0| + |0-1| = 4$

It seems that all the lines centering at $(0, 0)$ and in the space between points $(-1, 0)$ and $(-1, -1)$, and points $(1, 1)$ and $(1, 0)$ will have the same MAE, since the absolute difference of each pair of points' y values is always 1.

Meanwhile, since the origin never change, the absolute difference from point $(0, 1)$ and $(0, -1)$ to the point $(0, 0)$ is also always 1. The four points will have the total MAE 4 to the fitting line.

In a word, lines in the set that $0 \leqq slope \leqq 1$ and $intercept = 0$ all have the lowest MAE.

To further test our result, we implement a single-layer perceptron with linear function $(wx + b)$. Because the gradient of the MAE is a sign function, it's hard to derivate it manually, we use autograd function in Pytorch.

We set w (slope) >0, w <0 and w >1, and b (intercept) <0 and b >0, we find no matter which value w and b are, the optimal value set of the slope locates in (0,1], and meanwhile, the intercept always get a very small value around 0.

The code are shown as follows,

```
from torch.autograd import Variable
import torch

dtype = torch.FloatTensor
x = [-1,-1,0,0,0,1,1]
y = [-1,0,-1,0,1,0,1]
x_train = Variable(torch.tensor(x).type(dtype), requires_grad=True)
y_train = Variable(torch.tensor(y).type(dtype), requires_grad=True)

w = Variable(torch.tensor(-0.721).type(dtype), requires_grad=True)
b = Variable(torch.tensor(-1.472).type(dtype), requires_grad=True)

def linear_model(x):
        return w * x + b

optimizer = torch.optim.SGD([w, b], lr=0.001)
```

```
for i in range(20000):
y_ = linear_model(x_train)
loss = get_loss(y_, y_train)

optimizer.zero_grad()
loss.backward()
optimizer.step()

if i % 1000 == 0:
print('epoch: {}, loss: {}'.format(i, loss.data))
```

We have upload our Pytorch code to Colab, you can access the script at https://colab.research.google.com/drive/1P_Z6B-HPg8vtzVHL8EErORZ7uCY_9pZ0?usp=sharing, if you like.