Homework 1

Task 1: the Housing Prices

1. We download the train and the test dataset, and split the training dataset into train and validation sets.

Loading the train and test data.

In [2]:
```
train_df = pd.read_csv("train.csv", low_memory=False)
test_df = pd.read_csv("test.csv", low_memory=False)
train_df
```

Out[2]:

	Id	MSSubClass	MSZoning	LotFrontage	LotArea	Street	Alley	LotShape	LandContour	Utilities	...	PoolArea	PoolQC	Fence	MiscFeature	MiscV
0	1	60	RL	65.0	8450	Pave	NaN	Reg	Lvl	AllPub	...	0	NaN	NaN	NaN	
1	2	20	RL	80.0	9600	Pave	NaN	Reg	Lvl	AllPub	...	0	NaN	NaN	NaN	
2	3	60	RL	68.0	11250	Pave	NaN	IR1	Lvl	AllPub	...	0	NaN	NaN	NaN	
3	4	70	RL	60.0	9550	Pave	NaN	IR1	Lvl	AllPub	...	0	NaN	NaN	NaN	
4	5	60	RL	84.0	14260	Pave	NaN	IR1	Lvl	AllPub	...	0	NaN	NaN	NaN	
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	
1455	1456	60	RL	62.0	7917	Pave	NaN	Reg	Lvl	AllPub	...	0	NaN	NaN	NaN	
1456	1457	20	RL	85.0	13175	Pave	NaN	Reg	Lvl	AllPub	...	0	NaN	MnPrv	NaN	
1457	1458	70	RL	66.0	9042	Pave	NaN	Reg	Lvl	AllPub	...	0	NaN	GdPrv	Shed	25
1458	1459	20	RL	68.0	9717	Pave	NaN	Reg	Lvl	AllPub	...	0	NaN	NaN	NaN	
1459	1460	20	RL	75.0	9937	Pave	NaN	Reg	Lvl	AllPub	...	0	NaN	NaN	NaN	

1460 rows × 81 columns

In [3]:
```
from sklearn.model_selection import train_test_split


X = train_df.drop(columns = ['SalePrice']).copy()
y = train_df['SalePrice']

X_train, X_valid, y_train, y_valid = train_test_split(X,y, train_size = 0.8)


print(X_train.shape), print(y_train.shape)
print(X_valid.shape), print(y_valid.shape)
```

```
(1168, 80)
(1168,)
(292, 80)
(292,)
```

2. Examples of categorical features:

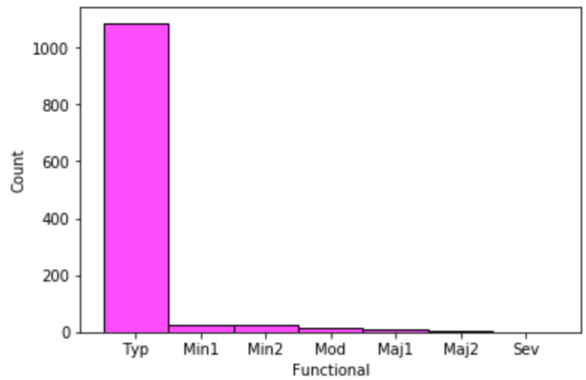
   - MSSubClass: The building class
   - Functional: Home functionality rating
   - Heating: Type of heating

   Examples of continuous features:

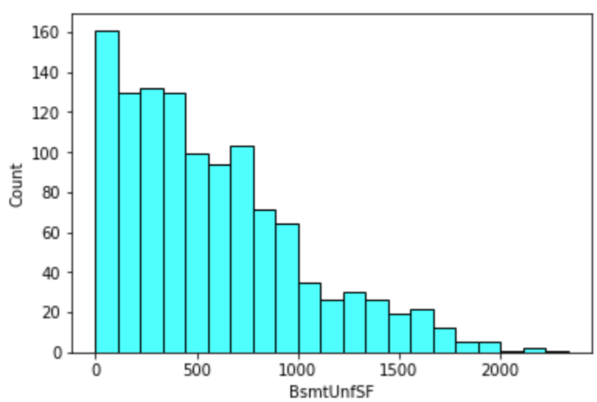
- 1stFlrSF: First Floor square feet
- GrLivArea: Above grade (ground) living area square feet
- SalePrice - the property's sale price in dollars

Plotting the Functional, BsmtUnfSF and SalePrice columns:

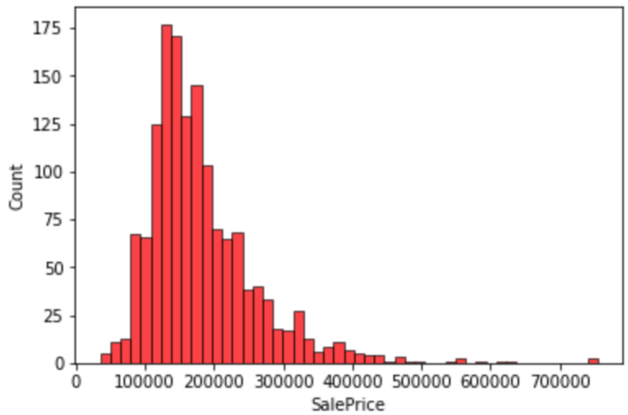
In [5]:
```
def plot_histogram(df, attr, colour):
    sns.histplot(data=df, x=attr, color=colour)

# Plotting the histogram for the attribute Functional
plot_histogram(X_train, "Functional", "magenta")
```




```
# Plotting the histogram for BsmtUnfSF
plot_histogram(X_train, "BsmtUnfSF", "cyan")
```



```
: # Plotting the histogram for Sale Price, the goal to be predicted
plot_histogram(train_df, "SalePrice", "red")
```



3. The preprocessing is a crucial step as we have a large dataset with 81 attributes. We want to handle noisy data, and also make smart choices about which attributes to pick or what features to engineer in order to train a model successfully. Firstly, we want to look at how many of the values are missing, and which attributes have the most missing values. We create a dataframe with attributes and the percentage of their missing values. We define a function that takes this information from the missing values dataframe and drops the columns that have more than 50% of its values missing.

In [8]:
```
# Creating a new empty dataframe
missing_df = pd.DataFrame()
missing_df["Feature"] = X_train.columns

# Calculating the percentage of the missing values for each attribute
missing = ((X_train.isnull().sum() / len(X_train)) * 100).values
missing_df["Missing"] = missing
missing_df = missing_df[missing_df["Feature"] != "SalePrice"]
missing_df = missing_df[missing_df["Missing"] != 0]
missing_df = missing_df.sort_values(by="Missing", ascending=False)

missing_df
```

Out[8]:

	Feature	Missing
72	PoolQC	99.400685
74	MiscFeature	95.976027
6	Alley	93.578767
73	Fence	80.650685
57	FireplaceQu	47.517123
3	LotFrontage	17.722603
58	GarageType	5.736301
59	GarageYrBlt	5.736301
60	GarageFinish	5.736301
63	GarageQual	5.736301
64	GarageCond	5.736301

In [9]:
```
# Defining a function to note down the columns to be removed, which have more than 50% of its values missing

def missing(df):

    attributes = df.loc[df['Missing'] > 50 ]

    return list(attributes['Feature'])

to_remove = missing(missing_df)

def remove_missing(df, to_remove):

    return df.drop(columns=to_remove)
```

We then separate numerical and categorical attributes into 2 different dataframes. We perform data pre-processing according to each attribute and its characteristics. Firstly, we plot all features to make sense of the data. Plotting numerical and categorical features, we get multiple plots, which are too long to include in this report. They can be found in the Jupyter notebooks.

Numerical Features

Looking at these visualisations, we can make assumptions about which attributes will be more useful, or which attributes we should normalise/modify/drop. Firstly, among the numerical features, we do not want to normalise attributes such as "Year". We can see from the plots that some of the attributes do not offer much information about the dataset because one their values are mostly the same. Therefore we can drop the following attributes:

- MasVnrArea
- BsmtFinSF2

We want to normalise attributes that do not have a Gaussian distribution. We will also normalise all attributes depicting area in square foot.

```
In [14]: to_normalise = ["MSSubClass", "BsmtFinSF1", "BsmtUnfSF", "TotalBsmtSF", "1stFlrSF", "2ndFlrSF", "LowQualFinSF", "GrL
         other_numerical = list(set(numerical_attr) - set(to_normalise))
         other_numerical

Out[14]: ['LotFrontage',
          'KitchenAbvGr',
          'Fireplaces',
          'BsmtHalfBath',
          'Id',
          'BedroomAbvGr',
          'OverallCond',
          'BsmtFullBath',
          'MiscVal',
          'BsmtFinSF2',
          'YearBuilt',
          'LotArea',
          'FullBath',
          'HalfBath',
          'GarageYrBlt',
          'YrSold',
          'MoSold',
          'TotRmsAbvGrd',
          'MasVnrArea',
          'YearRemodAdd',
          'OverallQual',
          'GarageCars']
```

Categorical Features

Features to drop:

- Utilities
- RoofStyle
- RoofMatl
- BsmtCond
- BsmtFinType2
- GarageCond
- GarageQual
- PavedDrive

We want to One-Hot Encode features that have common values, and features that can offer information about Sale Price. From the plots, we can see that features such as Neighborhood, CentralAir, PoolQC, LandSlope, BsmtQual fit these criteria, therefore we can one-hot encode them.

We already moved columns with more than 50% of its values missing. For the rest of the missing values, we replace them with the mean of that specific attribute using SimpleImputer.

In [16]:
```python
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import StandardScaler

# Using SimpleImputer to replace the missing values with the attributes' means
imputer = SimpleImputer(missing_values=np.nan, strategy='mean')
numerical_norm = imputer.fit_transform(numerical_norm)
numerical_rest = imputer.fit_transform(numerical_rest)


# To normalise the numerical attributes, we use StandardScaler
scaler = StandardScaler()
numerical_norm = scaler.fit_transform(numerical_norm)

numerical_norm = pd.DataFrame(numerical_norm, columns = to_normalise)
numerical_rest = pd.DataFrame(numerical_rest, columns = other_numerical )

numerical_norm
```

Out[16]:

	MSSubClass	BsmtFinSF1	BsmtUnfSF	TotalBsmtSF	1stFlrSF	2ndFlrSF	LowQualFinSF	GrLivArea	GarageArea	WoodDeckSF	OpenPorchSF	EnclosedPo
0	2.533660	-0.269860	-0.918919	-1.289458	-1.752866	0.369773	-0.120472	-1.003903	-0.911905	-0.765372	-0.713589	-0.351
1	2.533660	-0.956960	0.115825	-0.976170	-1.393773	0.645390	-0.120472	-0.511760	0.710684	-0.765372	0.084841	-0.351
2	3.019650	0.025528	-1.084299	-1.148258	-0.263388	-0.778253	-0.120472	-0.844845	0.240368	-0.765372	-0.299589	-0.351
3	1.561680	0.361586	0.332607	0.596890	0.409280	-0.778253	-0.120472	-0.347089	0.508448	-0.765372	-0.713589	2.464
4	-0.868270	-0.554546	-0.825055	-0.016449	-0.293733	-0.778253	-0.120472	-0.867300	-0.958936	2.620597	0.617128	-0.351
...	...	...	...	...	...	...	...	...	...	...	...	
1163	2.533660	-0.372604	-1.280968	-1.150464	-1.593550	0.465442	-0.120472	-0.807420	-0.883686	1.219506	-0.713589	-0.351
1164	0.832695	1.363339	-0.860813	0.451277	0.242378	1.335573	-0.120472	1.265940	-2.228788	-0.765372	0.321413	-0.351
1165	0.103710	0.273826	-0.192587	0.164464	-0.002919	2.729605	-0.120472	2.229641	1.679534	-0.765372	2.302702	-0.351
1166	2.533660	0.216033	-1.200513	-0.684944	-0.635125	0.474553	-0.120472	-0.090726	-0.648528	0.101970	-0.329160	-0.351
1167	-0.868270	-0.956960	1.483564	0.374058	0.194330	-0.778253	-0.120472	-0.506146	-0.338120	1.069389	-0.403089	-0.351

1168 rows × 15 columns

For the categorical attributes, we replace the missing values by the most frequent value in a column using SimpleImputer.

In [17]:
```python
# For the categorical attributes, we replace the missing values by the most frequent value in a column using Simple

imputer = SimpleImputer(missing_values=np.nan, strategy='most_frequent')
imputer = imputer.fit(df_categorical)

df_categorical = imputer.transform(df_categorical)
df_categorical = pd.DataFrame(df_categorical, columns = categorical_attr)

df_categorical
```

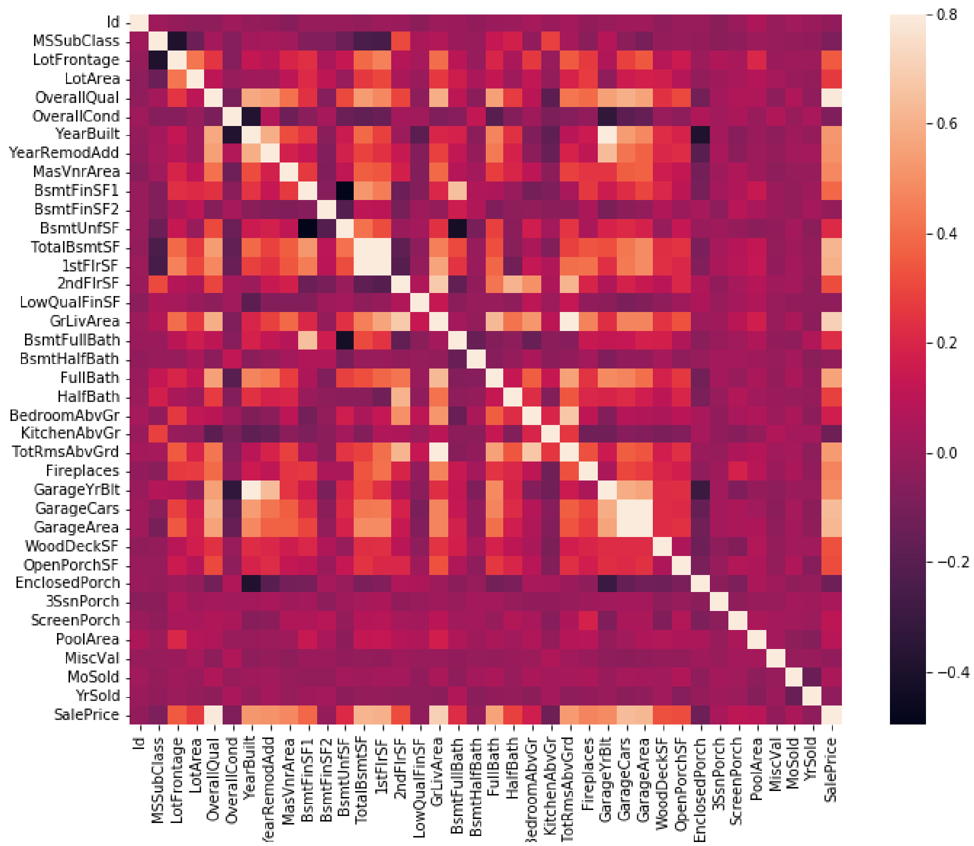
Out[17]:

	MSZoning	Street	Alley	LotShape	LandContour	Utilities	LotConfig	LandSlope	Neighborhood	Condition1	...	GarageType	GarageFinish	GarageQual
0	RM	Pave	Grvl	Reg	Lvl	AllPub	Inside	Gtl	BrDale	Norm	...	Detchd	Unf	TA
1	FV	Pave	Pave	IR1	Lvl	AllPub	Inside	Gtl	Somerst	Norm	...	Detchd	Fin	TA
2	RM	Pave	Grvl	Reg	Lvl	AllPub	Inside	Gtl	Edwards	Norm	...	Basment	RFn	TA
3	RL	Pave	Grvl	Reg	Lvl	AllPub	Inside	Gtl	StoneBr	Norm	...	Attchd	Fin	TA
4	RL	Pave	Grvl	IR1	Lvl	AllPub	Inside	Gtl	Sawyer	Norm	...	Attchd	Unf	TA
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
1163	RM	Pave	Grvl	Reg	Lvl	AllPub	Inside	Gtl	MeadowV	Norm	...	Attchd	RFn	TA
1164	RL	Pave	Grvl	Reg	Lvl	AllPub	Inside	Gtl	Edwards	Feedr	...	Attchd	Unf	TA
1165	RL	Pave	Grvl	IR1	Lvl	AllPub	Inside	Gtl	NWAmes	Norm	...	BuiltIn	Fin	TA
1166	RM	Pave	Grvl	Reg	Lvl	AllPub	Inside	Gtl	MeadowV	Norm	...	Attchd	RFn	TA
1167	RL	Pave	Grvl	Reg	Lvl	AllPub	Inside	Gtl	Gilbert	Norm	...	Attchd	Fin	TA

1168 rows × 43 columns

We merge the categorical and numerical dataframes into one.

We also want to plot the correlations of each attribute with SalePrice. We want to pick the attributes that will give us more information about predicting SalePrice. Plotting the correlation matrix, we get:

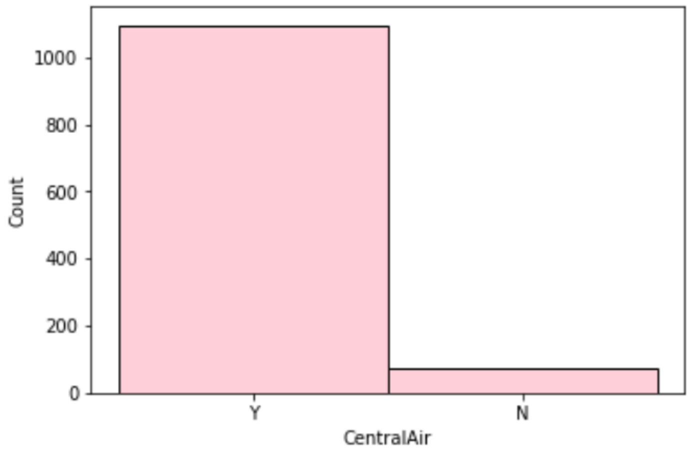


Using this information, we can drop the values that have a negative correlation and a correlation value lower than 0.3.

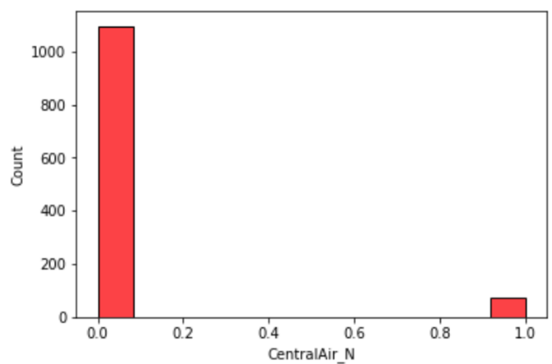
4. For One-Hot encoding, we want to pick categorical attributes that do not have too many distinct values, as they will be increasing the size of our training dataset drastically. We one-hot encode relevant features that might help with predicting SalePrice. Examples include BsmtQual and CentralAir.

   Here are the histplots for before and after one-hot encoding CentralAir.

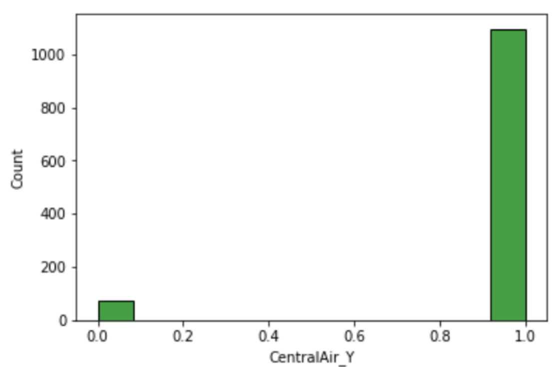
   Then, we Label Encode the rest of the categorical values.




```
In [25]: # Plotting the histogram for CentralAir after OHE
         plot_histogram(dum_df,"CentralAir_N", "red")
```




```
In [26]: plot_histogram(dum_df,"CentralAir_Y", "green")
```



5. Finally, when using Ordinary Least Squares for predicting house prices, we followed these steps to make an accurate feature selection:

(a) Dropped features with more than 50% of its values missing.

(b) Dropped numerical features that had little correlation with SalePrice (less than 30%).

(c) Dropped irrelevant or similar features (for example GarageCond and GarageQual).

We apply the ordinary least squares algorithm as demonstrated in the figure.

```
In [28]: theta_best = np.linalg.inv(encoded_df.T.dot(encoded_df)).dot(encoded_df.T).dot(y_train)
         theta_best_df = pd.DataFrame(data=theta_best[np.newaxis, :], columns=encoded_df.columns)
         theta_best_df
Out[28]:
```

	BsmtFinSF1	TotalBsmtSF	1stFlrSF	2ndFlrSF	GrLivArea	GarageArea	WoodDeckSF	OpenPorchSF	LotFrontage	Fireplaces	...	Electrical	KitchenQu
0	51.296195	36.218191	65.604691	156.718675	-4.02627	32.789575	46.680541	-6.972081	47.217525	8236.575238	...	-162.020143	-9094.0535

After preprocessing our validation and test data, we make predictions.

```
In [55]: valid_processed = preprocess(X_valid)
         test_processed = preprocess(test_df)
```

```
In [32]: # Generate predictions on the new prices
         y_valid_pred = valid_processed.dot(theta_best)
         y_test_pred = test_processed.dot(theta_best)

         y_test_pred

Out[32]: 0       116045.056802
         1       183226.499495
         2       188935.273259
         3       203624.869608
         4       200240.188527
                     ...
         1454     49196.617385
         1455     49786.906855
         1456    186743.961672
         1457    118115.459901
         1458    281044.019108
         Length: 1459, dtype: float64
```

Calculating the MSE and R^2, we get 0.48 and 2532109742.5 consecutively.

```
In [56]: from sklearn.metrics import r2_score
         from sklearn.metrics import mean_squared_error

         r2_score(y_valid, y_valid_pred)

Out[56]: 0.4817966893920168
```

```
In [57]: mean_squared_error(y_valid, y_valid_pred)
Out[57]: 2532109742.5113893
```

Task 2

1. We start by downloading the training and the test data.

```
In [2]: train_df = pd.read_csv("train.csv", low_memory=False)
        test_df = pd.read_csv("test.csv", low_memory=False)

        y_train = train_df["Survived"]
        train_df = train_df.drop(columns="Survived")
```

```
In [3]: train_df
```

Out[3]:

	PassengerId	Pclass	Name	Sex	Age	SibSp	Parch	Ticket	Fare	Cabin	Embarked
0	1	3	Braund, Mr. Owen Harris	male	22.0	1	0	A/5 21171	7.2500	NaN	S
1	2	1	Cumings, Mrs. John Bradley (Florence Briggs Th...	female	38.0	1	0	PC 17599	71.2833	C85	C
2	3	3	Heikkinen, Miss. Laina	female	26.0	0	0	STON/O2. 3101282	7.9250	NaN	S
3	4	1	Futrelle, Mrs. Jacques Heath (Lily May Peel)	female	35.0	1	0	113803	53.1000	C123	S
4	5	3	Allen, Mr. William Henry	male	35.0	0	0	373450	8.0500	NaN	S
...	...	...	...	...	...	...	...	...	...	...	...
886	887	2	Montvila, Rev. Juozas	male	27.0	0	0	211536	13.0000	NaN	S
887	888	1	Graham, Miss. Margaret Edith	female	19.0	0	0	112053	30.0000	B42	S
888	889	3	Johnston, Miss. Catherine Helen "Carrie"	female	NaN	1	2	W./C. 6607	23.4500	NaN	S
889	890	1	Behr, Mr. Karl Howell	male	26.0	0	0	111369	30.0000	C148	C
890	891	3	Dooley, Mr. Patrick	male	32.0	0	0	370376	7.7500	NaN	Q

891 rows × 11 columns

We then start preprocessing. We first eliminate attributes with missing values.

```
In [5]: # Removing missing values

        # Creating a new empty dataframe
        missing_df = pd.DataFrame()
        missing_df["Feature"] = train_df.columns

        # Calculating the percentage of the missing values for each attribute
        missing = ((train_df.isnull().sum() / len(train_df)) * 100).values
        missing_df["Missing"] = missing
        missing_df = missing_df[missing_df["Missing"] != 0]
        missing_df = missing_df.sort_values(by="Missing", ascending=False)

        missing_df
```

Out[5]:

	Feature	Missing
9	Cabin	77.104377
4	Age	19.865320
10	Embarked	0.224467

We can dropping columns that do not offer much information. Here, the ticket number and the cabin attributes do not offer information that might be relevant for our predictions.

We replace the missing values in columns Age and Embarked with their mean/most frequent value.

We then one-hot encode categorical attributes: Pclass, Sex and Embarked.

In [7]:
```python
def missing(df):

    attributes = df.loc[df['Missing'] > 50 ]

    return list(attributes['Feature'])

to_remove = missing(missing_df)

def remove_missing(df, to_remove):

    return df.drop(columns=to_remove)
```

In [6]:
```python
to_drop = ["Name", "Ticket"]
train_df = train_df.drop(columns=to_drop, axis=1)
```

In [9]:
```python
train_df['Age'] = train_df['Age'].fillna(train_df['Age'].mean())

train_df['Embarked'] = train_df['Embarked'].fillna(train_df['Embarked'].value_counts().idxmax())
train_df
```

Out[9]:

	PassengerId	Pclass	Sex	Age	SibSp	Parch	Fare	Embarked
0	1	3	male	22.000000	1	0	7.2500	S
1	2	1	female	38.000000	1	0	71.2833	C
2	3	3	female	26.000000	0	0	7.9250	S
3	4	1	female	35.000000	1	0	53.1000	S
4	5	3	male	35.000000	0	0	8.0500	S
...	...	...	...	...	...	...	...	...
886	887	2	male	27.000000	0	0	13.0000	S
887	888	1	female	19.000000	0	0	30.0000	S
888	889	3	female	29.699118	1	2	23.4500	S
889	890	1	male	26.000000	0	0	30.0000	C
890	891	3	male	32.000000	0	0	7.7500	Q

891 rows × 8 columns

In [10]:
```python
dummies = []
cols = ['Pclass', 'Sex', 'Embarked']
for col in cols:
    dummies.append(pd.get_dummies(train_df[col]))
```

In [11]:
```python
train_dummies = pd.concat(dummies, axis=1)
train_df = pd.concat((train_df,train_dummies), axis=1)
train_df = train_df.drop(columns=cols)

train_df
```

Out[11]:

	PassengerId	Age	SibSp	Parch	Fare	1	2	3	female	male	C	Q	S
0	1	22.000000	1	0	7.2500	0	0	1	0	1	0	0	1
1	2	38.000000	1	0	71.2833	1	0	0	1	0	1	0	0
2	3	26.000000	0	0	7.9250	0	0	1	1	0	0	0	1
3	4	35.000000	1	0	53.1000	1	0	0	1	0	0	0	1
4	5	35.000000	0	0	8.0500	0	0	1	0	1	0	0	1
...	...	...	...	...	...	...	...	...	...	...	...	...	...
886	887	27.000000	0	0	13.0000	0	1	0	0	1	0	0	1
887	888	19.000000	0	0	30.0000	1	0	0	1	0	0	0	1
888	889	29.699118	1	2	23.4500	0	0	1	1	0	0	0	1
889	890	26.000000	0	0	30.0000	1	0	0	0	1	1	0	0
890	891	32.000000	0	0	7.7500	0	0	1	0	1	0	1	0

891 rows × 13 columns

2. For this question, I implemented a logistic regression class from scratch. However, I used sklearn's logistic regression model for the Kaggle competition as it had a much higher accuracy. The code snippet shows both models.

   With the final model, we were able to get an accuracy of 0.805. Comparing this to Task

In [165]:
```
import numpy as np


class log_reg:
    """Logistic regression model"""

    def __init__(self, X):
        D = X.shape[1]
        self.w = np.zeros((D,1))

    def fit(self,X,y,alpha=0.01):
        w_star = self.grad_desc(X,y,alpha)
        self.w = w_star


    def logistic(logit):
        return (1/(1 + np.exp(-logit)))

    def gradient(X, y, w):
        N, D = X.shape
        yh = log_reg.logistic(np.dot(X, w))
        grad = np.dot(np.transpose(X), yh - y) / N
        return grad

    def grad_desc(self,
                  X,  # N x D
                  y,  # N
                  alpha,  # learning rate
                  eps=1e-2,  # termination condition
                  ):
        self.iters=0
        curr_w = self.w
        N, D = X.shape
        g = np.inf
        while np.linalg.norm(g) > eps:
            self.iters += 1
            if self.iters >= 200000 :
                return curr_w
            g = log_reg.gradient(X, y, curr_w)
            curr_w = curr_w - alpha*g
        return curr_w

    def predict(self,X):
        yh = log_reg.logistic(np.dot(X, self.w))
        rounder = lambda x: round(x)
        vfunc = np.vectorize(rounder)
        yh = vfunc(yh)
        return yh
```

In [166]:
```
def evaluate_acc(y,yhat):
    acc = 0.0
    for i in range(len(y)):
        if y[i] == yhat[i]:
            acc += 1.0
    return acc/len(y) #as a fraction
```

In [167]:
```
X_tr = train_df
Y_tr = y_train.to_numpy()


model = log_reg(X_tr)
model.fit(X_tr,Y_tr, alpha=0.01)
yhat_test = model.predict(X_test)
yhat_test
```

Out[167]:
```
array([[0., 1., 1., ..., 0., 1., 0.],
       [0., 1., 1., ..., 0., 1., 0.],
       [0., 1., 1., ..., 0., 1., 0.],
       ...,
       [0., 1., 1., ..., 0., 1., 0.],
       [0., 1., 1., ..., 0., 1., 0.],
       [0., 1., 1., ..., 0., 1., 0.]])
```

I, the accuracy on the Titanic dataset seems to be much higher using logistic regression.

In [19]:
```
from sklearn.linear_model import LogisticRegression

log_reg = LogisticRegression()
log_reg.fit(train_df, y_train)
log_reg.predict(train_df)

predictions = log_reg.predict(X_test)
```

In [20]:
```
pred_series = pd.Series(predictions)
pred_series
```

Out[20]:
```
0      0
1      0
2      0
3      0
4      1
      ..
413    0
414    1
415    0
416    0
417    0
Length: 418, dtype: int64
```

In [21]:
```
score = log_reg.score(train_df, y_train)
print(score)
```

```
0.8047138047138047
```

In [22]:
```
df = pd.concat([X_test['PassengerId'], pred_series], axis=1)
df = df.rename(columns={0: "Survived"})
df
```

Out[22]:

	PassengerId	Survived
0	892	0
1	893	0
2	894	0
3	895	0
4	896	1
...	...	...
413	1305	0
414	1306	1
415	1307	0
416	1308	0
417	1309	0

418 rows × 2 columns

Written Exercises

1. We start by writing the definition of the Kullback−Leibler (KL) divergence:

$$KL(p(x)||q(x)) = \mathbb{E}_{p(x)}\left[\log p(x) - \log q(x)\right]$$

Figure 1: Kullback−Leibler divergence

We want to prove that

We start with the definition of the KL divergence. Since we have a difference of two expected

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

values, we can make use of the linearity of expectation, apply the expected value to the right logarithm. Then, we can use Bayes' theorem as we have conditional probabilities. The equation we end up with is a function of θ , which allows us to maximise the second term independently of the first one.

$$\arg max_{\theta\in\Theta} \ \{E_{p(x,y)}[\log(p_\theta(y|x))]\}$$

$$= \ \arg min_{\theta\in\Theta} \ \left\{E_{p(x)} \left[E_{p(y|x)}[\log(p(y|x)) - \log(p_\theta(\mathrm{y|x}))]\right]\right\}$$

$$= \ \arg min_{\theta\in\Theta} \ \left\{E_{p(x)} \left[E_{p(y|x)}[\log(p(y|x))]\right] - \ E_{p(x)} \left[E_{p(y|x)}[\log(p_\theta(\mathrm{y|x}))]\right]\right\}$$

$$= \ \arg min_{\theta\in\Theta} \ \left\{\left[E_{p(x,y)}[\log(p(y|x))]\right] - \ \left[E_{p(x,y)}[\log(p_\theta(\mathrm{y|x}))]\right]\right\}$$

$$= \ \arg max_{\theta\in\Theta} \ \{E_{p(x,y)}[\log(p_\theta(\mathrm{y|x}))]\}$$

2. a) We have;

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

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

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

And we can substitute:

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

Which is

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

Therefore

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

b)

$$\frac{dL(\theta)}{d\theta} = -\frac{d}{d\theta} y log\sigma(\theta^T x) - \frac{d}{d\theta}(1-y)log[1-\sigma(\theta^T x)]$$

Taking the derivative of the sum of the terms, then taking deriving the logarithms:

$$[\frac{1-y}{\sigma(\theta^T x)} - \frac{y}{1-\sigma(\theta^T x)}]\frac{d}{d\theta}\sigma(\theta^T x)$$

Applying the chain rule, and the derivative of

$$\sigma$$

:

$$[\frac{1-y}{\sigma(\theta^T x)} - \frac{y}{1-\sigma(\theta^T x)}]\sigma(\theta^T x)[1-\sigma(\theta^T x)]x$$

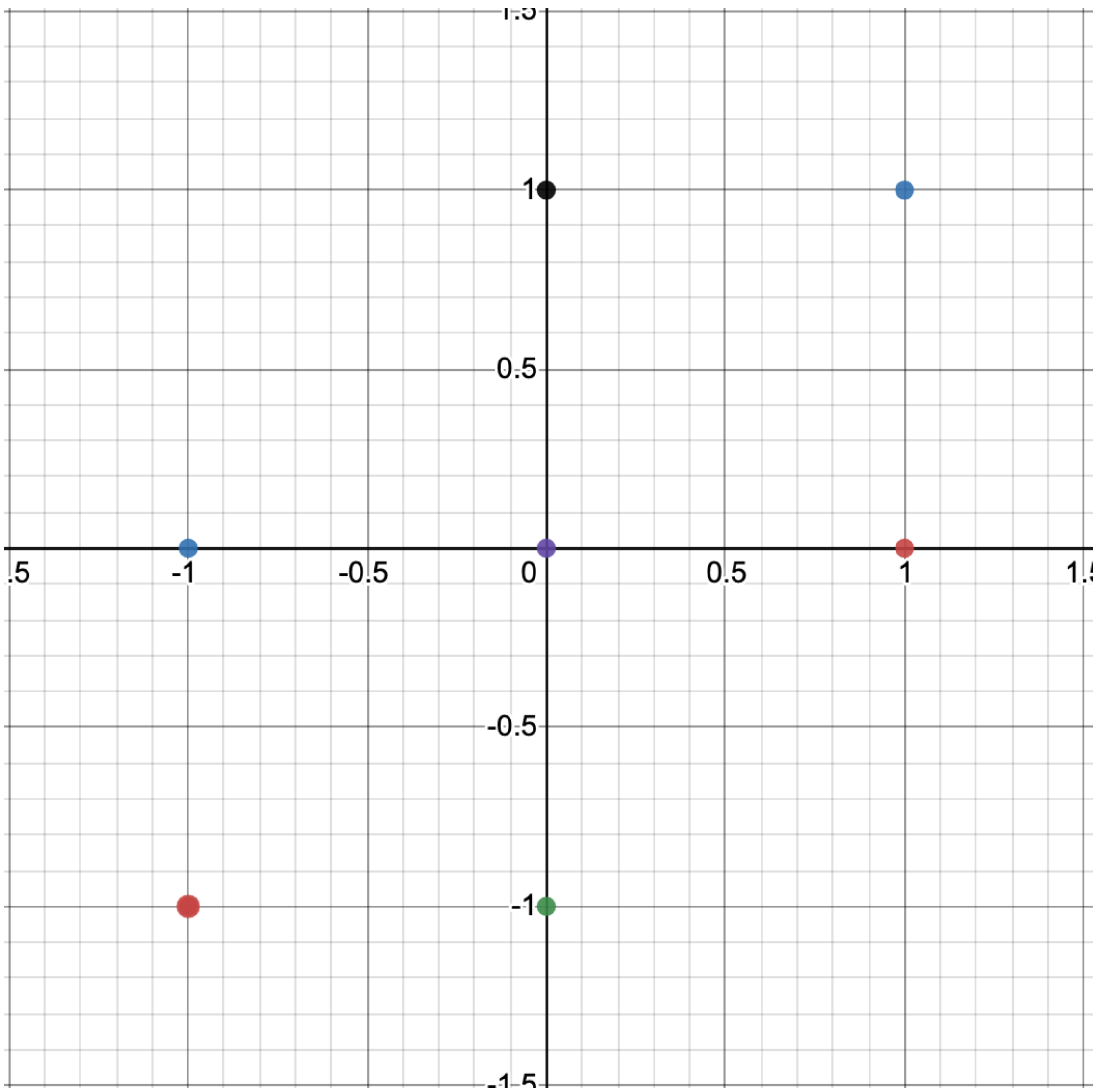
After moving the terms around we can cancel:

$$[\frac{\sigma(\theta^T x) - y}{\sigma(\theta^T x)[1-\sigma(\theta^T x)]}]\sigma(\theta^T x)[1-\sigma(\theta^T x)]x$$

We get:

$$[\sigma(\theta^T x) - y]x$$

3. a) Plotting the data points:



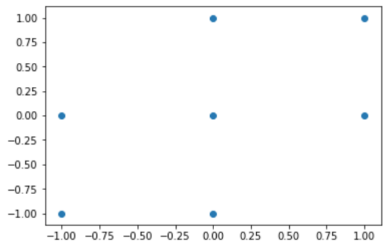
Without making any calculations, we expect the slope of the best fit line to be 0.5.

Now visualising this in Python, and calculating the linear regression we get the following:

We can see from the results that our initial guess was correct, and the slope of the best fit line is 0.5.

```
In [27]: import pandas as pd
         import matplotlib.pyplot as plt
         %matplotlib inline
```

```
In [29]: points = pd.DataFrame([[-1,-1], [-1, 0], [0, -1], [0, 0], [0, 1], [1, 0], [1, 1]])
         plt.scatter(points[0], points[1])

Out[29]: <matplotlib.collections.PathCollection at 0x7ff86af11b70>
```




```
In [31]: from scipy import stats

         col1 = points[0]
         col2 = points[1]

         slope, intercept, r, p, error = stats.linregress(col1, col2)

         best_fit = slope * col1 + intercept
```

```
In [32]: best_fit

Out[32]: 0   -0.5
         1   -0.5
         2    0.0
         3    0.0
         4    0.0
         5    0.5
         6    0.5
         Name: 0, dtype: float64
```

b&c) Now we calculate the MSE and the MAE using sklearn.

The reason why the slope of the best fit line is 0.5 is because the given points are symmetrical according to the

$$y = x$$

axis. The best fit line itself is this axis. It makes sense that the MSE and the MAE are relatively low, 0.14 and 0.29 respectively. There aren't any outliers, the distance of the data points to the best fit line are similar.

Mean Squared Error

```
In [44]: from sklearn.metrics import mean_squared_error

         results = best_fit.to_numpy()
         results

Out[44]: array([-0.5, -0.5,  0. ,  0. ,  0. ,  0.5,  0.5])
```

```
In [45]: mean_squared_error(col1, results)
Out[45]: 0.14285714285714285
```

Mean Absolute Error

```
In [46]: from sklearn.metrics import mean_absolute_error

         mean_absolute_error(col1, results)

Out[46]: 0.2857142857142857
```