Regression Setup, Train-test Split of the dataset

Introduction

We will be working with a data set based on housing prices in Ames, Iowa. It was compiled for educational use to be a modernized and expanded alternative to the well-known Boston Housing dataset. This version of the data set has had some missing values filled for convenience.

There are an extensive number of features, so they've been described in the table below.

Predictor

- SalePrice: The property's sale price in dollars.

Features

- MoSold: Month Sold
- YrSold: Year Sold
- SaleType: Type of sale
- SaleCondition: Condition of sale
- MSSubClass: The building class
- MSZoning: The general zoning classification
- ...

In [1]:
```
import os
data_path = ['data']
```

- Importing the data using Pandas and examine the shape. There are 79 feature columns plus the predictor, the sale price (`SalePrice`).
- There are three different types: integers (`int64`), floats (`float64`), and strings (`object` , categoricals).

In [3]:
```
import pandas as pd
import numpy as np

# Import the data using the file path
filepath = os.sep.join(data_path + ['Ames_Housing_Sales.csv']) #So our data_path is one list, we add that to the Ames_Housing_Sales.csv, which is another list, to have a list with
data = pd.read_csv("C:\\Users\\rsnen\\Desktop\\IBM\\KEh0FlFxQOuIdBZRcXDrwA_0dde15d9a6514d0e8a0fa13c41b9e1d7_Ames_Housing_Sales.csv", sep=',')

print(data.shape)
```
```
(1379, 80)
```

In [5]:
```
os.sep # gives seperater
```
Out[5]:
```
'\\'
```

In [7]:
```
data.dtypes.value_counts()
```
Out[7]:
```
object     43
float64    21
int64      16
dtype: int64
```

In [8]:
```
data.info()
```
```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1379 entries, 0 to 1378
Data columns (total 80 columns):
 #   Column         Non-Null Count  Dtype  
---  ------         --------------  -----  
 0   1stFlrSF       1379 non-null   float64
 1   2ndFlrSF       1379 non-null   float64
 2   3SsnPorch      1379 non-null   float64
 3   Alley          1379 non-null   object 
 4   BedroomAbvGr   1379 non-null   int64  
 5   BldgType       1379 non-null   object 
 6   BsmtCond       1379 non-null   object 
 7   BsmtExposure   1379 non-null   object 
 8   BsmtFinSF1     1379 non-null   float64
 9   BsmtFinSF2     1379 non-null   float64
 10  BsmtFinType1   1379 non-null   object 
 11  BsmtFinType2   1379 non-null   object 
 12  BsmtFullBath   1379 non-null   int64  
 13  BsmtHalfBath   1379 non-null   int64  
 14  BsmtQual       1379 non-null   object 
 15  BsmtUnfSF      1379 non-null   float64
 16  CentralAir     1379 non-null   object 
 17  Condition1     1379 non-null   object 
 18  Condition2     1379 non-null   object 
 19  Electrical     1379 non-null   object 
 20  EnclosedPorch  1379 non-null   float64
 21  ExterCond      1379 non-null   object 
 22  ExterQual      1379 non-null   object 
 23  Exterior1st    1379 non-null   object 
 24  Exterior2nd    1379 non-null   object 
 25  Fence          1379 non-null   object 
 26  FireplaceQu    1379 non-null   object 
 27  Fireplaces     1379 non-null   int64  
 28  Foundation     1379 non-null   object 
 29  FullBath       1379 non-null   int64  
 30  Functional     1379 non-null   object 
 31  GarageArea     1379 non-null   float64
 32  GarageCars     1379 non-null   int64  
 33  GarageCond     1379 non-null   object 
 34  GarageFinish   1379 non-null   object 
 35  GarageQual     1379 non-null   object 
 36  GarageType     1379 non-null   object 
 37  GarageYrBlt    1379 non-null   float64
 38  GrLivArea      1379 non-null   float64
 39  HalfBath       1379 non-null   int64  
 40  Heating        1379 non-null   object 
 41  HeatingQC      1379 non-null   object 
 42  HouseStyle     1379 non-null   object 
 43  KitchenAbvGr   1379 non-null   int64  
 44  KitchenQual    1379 non-null   object 
 45  LandContour    1379 non-null   object 
 46  LandSlope      1379 non-null   object 
 47  LotArea        1379 non-null   float64
 48  LotConfig      1379 non-null   object 
 49  LotFrontage    1379 non-null   float64
 50  LotShape       1379 non-null   object 
 51  LowQualFinSF   1379 non-null   float64
 52  MSSubClass     1379 non-null   int64  
 53  MSZoning       1379 non-null   object 
 54  MasVnrArea     1379 non-null   float64
 55  MasVnrType     1379 non-null   object 
 56  MiscFeature    1379 non-null   object 
 57  MiscVal        1379 non-null   float64
 58  MoSold         1379 non-null   int64  
 59  Neighborhood   1379 non-null   object 
 60  OpenPorchSF    1379 non-null   float64
 61  OverallCond    1379 non-null   int64  
 62  OverallQual    1379 non-null   int64  
 63  PavedDrive     1379 non-null   object 
 64  PoolArea       1379 non-null   float64
 65  PoolQC         1379 non-null   object 
 66  RoofMatl       1379 non-null   object 
 67  RoofStyle      1379 non-null   object 
 68  SaleCondition  1379 non-null   object 
 69  SaleType       1379 non-null   object 
 70  ScreenPorch    1379 non-null   float64
 71  Street         1379 non-null   object 
 72  TotRmsAbvGrd   1379 non-null   int64  
 73  TotalBsmtSF    1379 non-null   float64
 74  Utilities      1379 non-null   object 
 75  WoodDeckSF     1379 non-null   float64
 76  YearBuilt      1379 non-null   int64  
 77  YearRemodAdd   1379 non-null   int64  
 78  YrSold         1379 non-null   int64  
 79  SalePrice      1379 non-null   float64
dtypes: float64(21), int64(16), object(43)
memory usage: 862.0+ KB
```

There is a significant challenge, particularly when dealing with data that have many columns, is ensuring each column gets encoded correctly.

This is particularly true with data columns that are ordered categoricals (ordinals) vs unordered categoricals. Unordered categoricals should be one-hot encoded, however this can significantly increase the number of features and creates features that are highly correlated with each other.

Determining how many total features would be present, relative to what currently exists, if all string (object) features are one-hot encoded. Remember total number of one-hot encoded columns is `n-1` , where `n` is the number of categories.

In [11]:
```
# Select the object (string) columns
mask = data.dtypes == np.object
categorical_cols = data.columns[mask]
```

In [12]:
```
mask #  mask is just going to be true or false Pandas series, that is the same length as your columns, in the order to filter down to the columns that are just equal to np.objects.
```
Out[12]:
```
1stFlrSF        False
2ndFlrSF        False
3SsnPorch       False
Alley            True
BedroomAbvGr    False
                ...  
WoodDeckSF      False
YearBuilt       False
YearRemodAdd    False
YrSold          False
SalePrice       False
Length: 80, dtype: bool
```

In [13]:
```
categorical_cols # We are going to filter down our data to just those columns(object datatype columns)
```
Out[13]:
```
Index(['Alley', 'BldgType', 'BsmtCond', 'BsmtExposure', 'BsmtFinType1',
       'BsmtFinType2', 'BsmtQual', 'CentralAir', 'Condition1', 'Condition2',
       'Electrical', 'ExterCond', 'ExterQual', 'Exterior1st', 'Exterior2nd',
       'Fence', 'FireplaceQu', 'Foundation', 'Functional', 'GarageCond',
       'GarageFinish', 'GarageQual', 'GarageType', 'Heating', 'HeatingQC',
       'HouseStyle', 'KitchenQual', 'LandContour', 'LandSlope', 'LotConfig',
       'LotShape', 'MSZoning', 'MasVnrType', 'MiscFeature', 'Neighborhood',
       'PavedDrive', 'PoolQC', 'RoofMatl', 'RoofStyle', 'SaleCondition',
       'SaleType', 'Street', 'Utilities'],
      dtype='object')
```

In [16]:
```
data[categorical_cols]
```
Out[16]:

	Alley	BldgType	BsmtCond	BsmtExposure	BsmtFinType1	BsmtFinType2	BsmtQual	CentralAir	Condition1	Condition2	...	MiscFeature	Neighborhood	PavedDrive	PoolQC	RoofMatl	RoofStyle	SaleCondition	SaleType
0	None	1Fam	TA	No	GLQ	Unf	Gd	Y	Norm	Norm	...	None	CollgCr	Y	None	CompShg	Gable	Normal	WD
1	None	1Fam	TA	Gd	ALQ	Unf	Gd	Y	Feedr	Norm	...	None	Veenker	Y	None	CompShg	Gable	Normal	WD
2	None	1Fam	TA	Mn	GLQ	Unf	Gd	Y	Norm	Norm	...	None	CollgCr	Y	None	CompShg	Gable	Normal	WD
3	None	1Fam	Gd	No	ALQ	Unf	TA	Y	Norm	Norm	...	None	Crawfor	Y	None	CompShg	Gable	Abnorml	WD
4	None	1Fam	TA	Av	GLQ	Unf	Gd	Y	Norm	Norm	...	None	NoRidge	Y	None	CompShg	Gable	Normal	WD
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
1374	None	1Fam	None	None	None	None	None	Y	Norm	Norm	...	None	Gilbert	Y	None	CompShg	Gable	Normal	WD
1375	None	1Fam	TA	No	ALQ	Rec	Gd	Y	Norm	Norm	...	None	NWAmes	Y	None	CompShg	Gable	Normal	WD
1376	None	1Fam	Gd	No	GLQ	Unf	TA	Y	Norm	Norm	...	Shed	Crawfor	Y	None	CompShg	Gable	Normal	WD
1377	None	1Fam	TA	Mn	GLQ	Rec	TA	Y	Norm	Norm	...	None	NAmes	Y	None	CompShg	Hip	Normal	WD
1378	None	1Fam	TA	No	BLQ	LwQ	TA	Y	Norm	Norm	...	None	Edwards	Y	None	CompShg	Gable	Normal	WD

1379 rows × 43 columns

In [17]:
```
data[categorical_cols].apply(lambda x: x.nunique()) # number of unique values in each column
```
Out[17]:
```
Alley             3
BldgType          5
BsmtCond          4
BsmtExposure      5
BsmtFinType1      6
BsmtFinType2      7
BsmtQual          5
CentralAir        2
Condition1        9
Condition2        8
Electrical        5
ExterCond         4
ExterQual         4
Exterior1st      14
Exterior2nd      16
Fence             5
FireplaceQu       6
Foundation        6
Functional        7
GarageCond        5
GarageFinish      3
GarageQual        5
GarageType        6
Heating           6
HeatingQC         5
HouseStyle        8
KitchenQual       4
LandContour       4
LandSlope         3
LotConfig         5
LotShape          4
MSZoning          5
MasVnrType        4
MiscFeature       5
Neighborhood     25
PavedDrive        3
PoolQC            4
RoofMatl          8
RoofStyle         6
SaleCondition     6
SaleType          9
Street            2
Utilities         2
dtype: int64
```

In [19]:
```
# Determine how many extra columns would be created
num_ohc_cols = (data[categorical_cols] #That's going to be our num_ohc_cols.
                .apply(lambda x: x.nunique())
                .sort_values(ascending=False))


# No need to encode if there is only one value
small_num_ohc_cols = num_ohc_cols.loc[num_ohc_cols>1] # locating any of those that are equal to 1 and filter those out using this.loc.

# Number of one-hot columns is one less than the number of categories
small_num_ohc_cols -= 1

# This is 215 columns, assuming the original ones are dropped.
# This is quite a few extra columns!
small_num_ohc_cols.sum()
```
Out[19]:
```
215
```

Creating a new data set where all of the above categorical features will be one-hot encoded. We can fit this data and see how it affects the results.

- Used the dataframe `.copy()` method to create a completely separate copy of the dataframe for one-hot encoding
- On this new dataframe, one-hot encode each of the appropriate columns and add it back to the dataframe. Be sure to drop the original column.
- For the data that are not one-hot encoded, drop the columns that are string categoricals.

For the first step, numerically encoding the string categoricals, either Scikit-learn;s `LabelEncoder` or `DictVectorizer` can be used. However, the former is probably easier since it doesn't require specifying a numerical value for each category, and we are going to one-hot encode all of the numerical values anyway.

In [20]:
```
from sklearn.preprocessing import OneHotEncoder, LabelEncoder

# Copy of the data
data_ohc = data.copy()

# The encoders
le = LabelEncoder()
ohc = OneHotEncoder()

for col in num_ohc_cols.index:
    
    # Integer encode the string categories
    dat = le.fit_transform(data_ohc[col]).astype(np.int)
    
    # Remove the original column from the dataframe
    data_ohc = data_ohc.drop(col, axis=1)

    # One hot encode the data--this returns a sparse array
    new_dat = ohc.fit_transform(dat.reshape(-1,1))

    # Create unique column names
    n_cols = new_dat.shape[1]
    col_names = ['_'.join([col, str(x)]) for x in range(n_cols)]

    # Create the new dataframe
    new_df = pd.DataFrame(new_dat.toarray(), 
                          index=data_ohc.index, 
                          columns=col_names)

    # Append the new data to the dataframe
    data_ohc = pd.concat([data_ohc, new_df], axis=1)
```

In [21]:
```
# Column difference is as calculated above
data_ohc.shape[1] - data.shape[1]
```
Out[21]:
```
215
```

In [22]:
```
print(data.shape[1])

# Remove the string columns from the dataframe
data = data.drop(num_ohc_cols.index, axis=1)

print(data.shape[1])
```
```
80
37
```

- Creating train and test splits of both data sets. To ensure the data gets split the same way, use the same `random_state` in each of the two splits.
- For each data set, fit a basic linear regression model on the training data.
- Calculate the mean squared error on both the train and test sets for the respective models. Which model produces smaller error on the test data and why?

In [23]:
```
#demo
from sklearn.model_selection import train_test_split

y_col = 'SalePrice'

# Split the data that is not one-hot encoded
feature_cols = [x for x in data.columns if x != y_col]
X_data = data[feature_cols]
y_data = data[y_col]

X_train, X_test, y_train, y_test = train_test_split(X_data, y_data, 
                                                    test_size=0.3, random_state=42)
# Split the data that is one-hot encoded
feature_cols = [x for x in data_ohc.columns if x != y_col]
X_data_ohc = data_ohc[feature_cols]
y_data_ohc = data_ohc[y_col]

X_train_ohc, X_test_ohc, y_train_ohc, y_test_ohc = train_test_split(X_data_ohc, y_data_ohc, 
                                                    test_size=0.3, random_state=42)
```

In [24]:
```
from sklearn.model_selection import train_test_split

y_col = 'SalePrice'

# Split the data that is not one-hot encoded
feature_cols = [x for x in data.columns if x != y_col]
X_data = data[feature_cols]
y_data = data[y_col]

X_train, X_test, y_train, y_test = train_test_split(X_data, y_data, 
                                                    test_size=0.3, random_state=42)
```

In [26]:
```
X_train.shape
```
Out[26]:
```
(965, 36)
```

In [27]:
```
y_train.shape
```
Out[27]:
```
(965,)
```

In [28]:
```
# Split the data that is one-hot encoded
feature_cols = [x for x in data_ohc.columns if x != y_col]
X_data_ohc = data_ohc[feature_cols]
y_data_ohc = data_ohc[y_col]

X_train_ohc, X_test_ohc, y_train_ohc, y_test_ohc = train_test_split(X_data_ohc, y_data_ohc, 
                                                    test_size=0.3, random_state=42)
```

In [29]:
```
X_train_ohc.shape
```
Out[29]:
```
(965, 294)
```

In [30]:
```
y_train_ohc.shape
```
Out[30]:
```
(965,)
```

In [33]:
```
# Compare the indices to ensure they are identical
(X_train_ohc.index == X_train.index).all() # .all funtion gives output true to every single value in that array has to be true for this to evaluate to true.
```
Out[33]:
```
True
```

In [34]:
```
(X_train_ohc.index == X_train.index)
```
Out[34]:
```
array([ True,  True,  True,  True,  True,  True,  True,  True,  True,
        True,  True,  True,  True,  True,  True,  True,  True,  True,
        True,  True,  True,  True,  True,  True,  True,  True,  True,
        True,  True,  True,  True,  True,  True,  True,  True,  True,
        True,  True,  True,  True,  True,  True,  True,  True,  True,
        True,  True,  True,  True,  True,  True,  True,  True,  True,
        True,  True,  True,  True,  True,  True,  True,  True,  True,
        True,  True,  True,  True,  True,  True,  True,  True,  True,
        True,  True,  True,  True,  True,  True,  True,  True,  True,
        True,  True,  True,  True,  True,  True,  True,  True,  True,
        True,  True,  True,  True,  True,  True,  True,  True,  True,
        True,  True,  True,  True,  True,  True,  True,  True,  True,
        True,  True,  True,  True,  True,  True,  True,  True,  True,
        True,  True,  True,  True,  True,  True,  True,  True,  True,
        True,  True,  True,  True,  True,  True,  True,  True,  True,
        True,  True,  True,  True,  True,  True,  True,  True,  True,
        True,  True,  True,  True,  True,  True,  True,  True,  True,
        True,  True,  True,  True,  True,  True,  True,  True,  True,
        True,  True,  True,  True,  True,  True,  True,  True,  True,
        True,  True,  True,  True,  True,  True,  True,  True,  True,
        True,  True,  True,  True,  True,  True,  True,  True,  True,
        True,  True,  True,  True,  True,  True,  True,  True,  True,
        True,  True,  True,  True,  True,  True,  True,  True,  True,
        True,  True,  True,  True,  True,  True,  True,  True,  True,
        True,  True,  True,  True,  True,  True,  True,  True,  True,
        True,  True,  True,  True,  True,  True,  True,  True,  True,
        True,  True,  True,  True,  True,  True,  True,  True,  True,
        True,  True,  True,  True,  True,  True,  True,  True,  True,
        True,  True,  True,  True,  True,  True,  True,  True,  True,
        True,  True,  True,  True,  True,  True,  True,  True,  True,
        True,  True,  True,  True,  True,  True,  True,  True,  True,
        True,  True,  True,  True,  True,  True,  True,  True,  True,
        True,  True,  True,  True,  True,  True,  True,  True,  True,
        True,  True,  True,  True,  True,  True,  True,  True,  True,
        True,  True,  True,  True,  True,  True,  True,  True,  True,
        True,  True,  True,  True,  True,  True,  True,  True,  True,
        True,  True,  True,  True,  True,  True,  True,  True,  True,
        True,  True,  True,  True,  True,  True,  True,  True,  True,
        True,  True,  True,  True,  True,  True,  True,  True,  True,
        True,  True,  True,  True,  True,  True,  True,  True,  True,
        True,  True,  True,  True,  True,  True,  True,  True,  True,
        True,  True,  True,  True,  True,  True,  True,  True,  True,
        True,  True,  True,  True,  True,  True,  True,  True,  True,
        True,  True,  True,  True,  True,  True,  True,  True,  True,
        True,  True,  True,  True,  True,  True,  True,  True,  True,
        True,  True,  True,  True,  True,  True,  True,  True,  True,
        True,  True,  True,  True,  True,  True,  True,  True,  True,
        True,  True,  True,  True,  True,  True,  True,  True,  True,
        True,  True,  True,  True,  True,  True,  True,  True,  True,
        True,  True,  True,  True,  True,  True,  True,  True,  True,
        True,  True,  True,  True,  True,  True,  True,  True,  True,
        True,  True,  True,  True,  True,  True,  True,  True,  True,
        True,  True,  True,  True,  True,  True,  True,  True,  True,
        True,  True,  True,  True,  True,  True,  True,  True,  True,
        True,  True,  True,  True,  True,  True,  True,  True,  True,
        True,  True,  True,  True,  True,  True,  True,  True,  True,
        True,  True,  True,  True,  True,  True,  True,  True,  True,
        True,  True,  True,  True,  True,  True,  True,  True,  True,
        True,  True,  True,  True,  True,  True,  True,  True,  True,
        True,  True,  True,  True,  True,  True,  True,  True,  True,
        True,  True,  True,  True,  True,  True,  True,  True,  True,
        True,  True,  True,  True,  True,  True,  True,  True,  True,
        True,  True,  True,  True,  True,  True,  True,  True,  True,
        True,  True,  True,  True,  True,  True,  True,  True,  True,
        True,  True,  True,  True,  True,  True,  True,  True,  True,
        True,  True,  True,  True,  True,  True,  True,  True,  True,
        True,  True,  True,  True,  True,  True,  True,  True,  True,
        True,  True,  True,  True,  True,  True,  True,  True,  True,
        True,  True,  True,  True,  True,  True,  True,  True,  True,
        True,  True,  True,  True,  True,  True,  True,  True,  True,
        True,  True,  True,  True,  True,  True,  True,  True,  True,
        True,  True,  True,  True,  True,  True,  True,  True,  True,
        True,  True,  True,  True,  True,  True,  True,  True,  True,
        True,  True,  True,  True,  True,  True,  True,  True,  True,
        True,  True,  True,  True,  True,  True,  True,  True,  True,
        True,  True,  True,  True,  True,  True,  True,  True,  True,
        True,  True,  True,  True,  True,  True,  True,  True,  True,
        True,  True,  True,  True,  True,  True,  True,  True,  True,
        True,  True,  True,  True,  True,  True,  True,  True,  True,
        True,  True,  True,  True,  True,  True,  True,  True,  True,
        True,  True,  True,  True,  True,  True,  True,  True,  True,
        True,  True,  True,  True,  True,  True,  True,  True,  True,
        True,  True,  True,  True,  True,  True,  True,  True,  True,
        True,  True,  True,  True,  True,  True,  True,  True,  True,
        True,  True,  True,  True,  True,  True,  True,  True,  True,
        True,  True,  True,  True,  True,  True,  True,  True,  True,
        True,  True,  True,  True,  True,  True,  True,  True,  True,
        True,  True,  True,  True,  True,  True,  True,  True,  True,
        True,  True,  True,  True,  True,  True,  True,  True,  True,
        True,  True,  True,  True,  True,  True,  True,  True,  True,
        True,  True,  True,  True,  True,  True,  True,  True,  True,
        True,  True,  True,  True,  True,  True,  True,  True,  True,
        True,  True,  True,  True,  True,  True,  True,  True,  True,
        True,  True,  True,  True,  True,  True,  True,  True,  True,
        True,  True,  True,  True,  True,  True,  True,  True,  True,
        True,  True,  True,  True,  True,  True,  True,  True,  True,
        True,  True,  True,  True,  True,  True,  True,  True,  True,
        True,  True,  True,  True,  True,  True,  True,  True,  True,
        True,  True,  True,  True,  True,  True,  True,  True,  True,
        True,  True,  True,  True,  True,  True,  True,  True,  True,
        True,  True,  True,  True,  True,  True,  True,  True,  True,
        True,  True,  True,  True,  True,  True,  True,  True,  True,
        True,  True,  True,  True,  True,  True,  True,  True,  True,
        True,  True,  True,  True,  True,  True,  True,  True,  True,
        True,  True,  True,  True,  True,  True,  True,  True,  True,
        True,  True,  True,  True,  True,  True,  True,  True,  True,
        True,  True,  True,  True,  True,  True,  True,  True,  True,
        True,  True,  True,  True,  True,  True,  True,  True,  True,
        True,  True,  True,  True,  True,  True,  True,  True,  True,
        True,  True,  True,  True,  True,  True,  True,  True,  True,
        True,  True,  True,  True,  True,  True,  True,  True,  True,
        True,  True,  True,  True,  True,  True,  True,  True,  True,
        True,  True,  True,  True,  True,  True,  True,  True,  True,
        True,  True,  True,  True,  True,  True,  True,  True,  True,
        True,  True,  True,  True,  True,  True,  True,  True,  True,
        True,  True,  True,  True,  True,  True,  True,  True,  True,
        True,  True,  True,  True,  True,  True,  True,  True,  True,
        True,  True,  True,  True,  True,  True,  True,  True,  True,
        True,  True,  True,  True,  True,  True,  True,  True,  True,
        True,  True,  True,  True,  True,  True,  True,  True,  True,
        True,  True,  True,  True,  True,  True,  True,  True,  True,
        True,  True])
```

In [35]:
```
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error

LR = LinearRegression()

# Storage for error values
error_df = list()

# Data that have not been one-hot encoded
LR = LR.fit(X_train, y_train)
y_train_pred = LR.predict(X_train)
y_test_pred = LR.predict(X_test)

error_df.append(pd.Series({'train': mean_squared_error(y_train, y_train_pred),
                           'test' : mean_squared_error(y_test,  y_test_pred)},
                           name='no enc'))

# Data that have been one-hot encoded
LR = LR.fit(X_train_ohc, y_train_ohc)
y_train_ohc_pred = LR.predict(X_train_ohc)
y_test_ohc_pred = LR.predict(X_test_ohc)

error_df.append(pd.Series({'train': mean_squared_error(y_train_ohc, y_train_ohc_pred),
                           'test' : mean_squared_error(y_test_ohc,  y_test_ohc_pred)},
                          name='one-hot enc'))

# Assemble the results
error_df = pd.concat(error_df, axis=1)
error_df
```
Out[35]:

	no enc	one-hot enc
train	1.131507e+09	3.177303e+08
test	1.372182e+09	3.180592e+19

Note that the error values on the one-hot encoded data are very different for the train and test data. In particular, the errors on the test data are much higher. Based on the lecture, this is because the one-hot encoded model is overfitting the data. We will learn how to deal with issues like this in the next lesson.

For each of the data sets (one-hot encoded and not encoded):

- Scaling the all the non-hot encoded values using one of the following: `StandardScaler` , `MinMaxScaler` , `MaxAbsScaler` .
- Comparing the error calculated on the test sets

Being sure to calculate the skew (to decide if a transformation should be done) and fit the scaler on *ONLY* the training data, but then apply it to both the train and test data identically.

In [36]:
```
# Mute the setting wtih a copy warnings
pd.options.mode.chained_assignment = None
```

In [37]:
```
from sklearn.preprocessing import StandardScaler, MinMaxScaler, MaxAbsScaler


scalers = {'standard': StandardScaler(),
           'minmax': MinMaxScaler(),
           'maxabs': MaxAbsScaler()}
scalers.items()
```
Out[37]:
```
dict_items([('standard', StandardScaler()), ('minmax', MinMaxScaler()), ('maxabs', MaxAbsScaler())])
```

In [40]:
```
from sklearn.preprocessing import StandardScaler, MinMaxScaler, MaxAbsScaler


scalers = {'standard': StandardScaler(),
           'minmax': MinMaxScaler(),
           'maxabs': MaxAbsScaler()}

training_test_sets = {
    'not_encoded': (X_train, y_train, X_test, y_test),
    'one_hot_encoded': (X_train_ohc, y_train_ohc, X_test_ohc, y_test_ohc)}


# Get the list of float columns, and the float data
# so that we don't scale something we already scaled.
# We're supposed to scale the original data each time
mask = X_train.dtypes == np.float
float_columns = X_train.columns[mask]

# initialize model
LR = LinearRegression()

# iterate over all possible combinations and get the errors
errors = {}
for encoding_label, (_X_train, _y_train, _X_test, _y_test) in training_test_sets.items():
    for scaler_label, scaler in scalers.items():
        trainingset = _X_train.copy()  # copy because we dont want to scale this more than once.
        testset = _X_test.copy()
        trainingset[float_columns] = scaler.fit_transform(trainingset[float_columns])
        testset[float_columns] = scaler.transform(testset[float_columns])
        LR.fit(trainingset, _y_train)
        predictions = LR.predict(testset)
        key = encoding_label + ' - ' + scaler_label + 'scaling'
        errors[key] = mean_squared_error(_y_test, predictions)

errors = pd.Series(errors)
print(errors.to_string())
print('-' * 80)
for key, error_val in errors.items():
    print(key, error_val)
```
```
not_encoded - standardscaling        1.372182e+09
not_encoded - minmaxscaling          1.372029e+09
not_encoded - maxabsscaling          1.371663e+09
one_hot_encoded - standardscaling    8.065328e+09
one_hot_encoded - minmaxscaling      8.065328e+09
one_hot_encoded - maxabsscaling      8.065328e+09
--------------------------------------------------------------------------------
not_encoded - standardscaling 1372182358.934503
not_encoded - minmaxscaling 1372029420.6916711
not_encoded - maxabsscaling 1371662758.7975926
one_hot_encoded - standardscaling 8065327607.247173
one_hot_encoded - minmaxscaling 8065327607.328744
one_hot_encoded - maxabsscaling 8065327607.082172
```

Plotting predictions vs actual for one of the models.

In [41]:
```
import matplotlib.pyplot as plt
import seaborn as sns
%matplotlib inline


sns.set_context('talk')
sns.set_style('ticks')
sns.set_palette('dark')

ax = plt.axes()
# we are going to use y_test, y_test_pred
ax.scatter(y_test, y_test_pred, alpha=.5)

ax.set(xlabel='Ground truth', 
       ylabel='Predictions',
       title='Ames, Iowa House Price Predictions vs Truth, using Linear Regression');
```