Introduction

We will begin with a short tutorial on regression, polynomial features, and regularization based on a very simple, sparse data set that contains a column of `x` data and associated `y` noisy data. The data file is called `X_Y_Sinusoid_Data.csv`.

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

- Importing the data.
- Also generate approximately 100 equally spaced x data points over the range of 0 to 1. Using these points, calculate the y-data which represents the "ground truth" (the real function) from the equation: $y = sin(2\pi x)$
- Plot the sparse data (`x` vs `y`) and the calculated ("real") data.

In [6]:
```
import pandas as pd
import numpy as np

filepath = os.sep.join(data_path + ['X_Y_Sinusoid_Data.csv'])
data = pd.read_csv("C:\\Users\\rsnen\\Desktop\\IBM\\_X_Y_Sinusoid_Data.csv")

X_real = np.linspace(0, 1.0, 100)
Y_real = np.sin(2 * np.pi * X_real)
```

In [9]:
```
data.head()
```

Out[9]:

	x	y
0	0.038571	0.066391
1	0.166776	1.027483
2	0.183153	1.245302
3	0.187359	1.004781
4	0.243116	1.264121

In [10]:
```
X_real
```

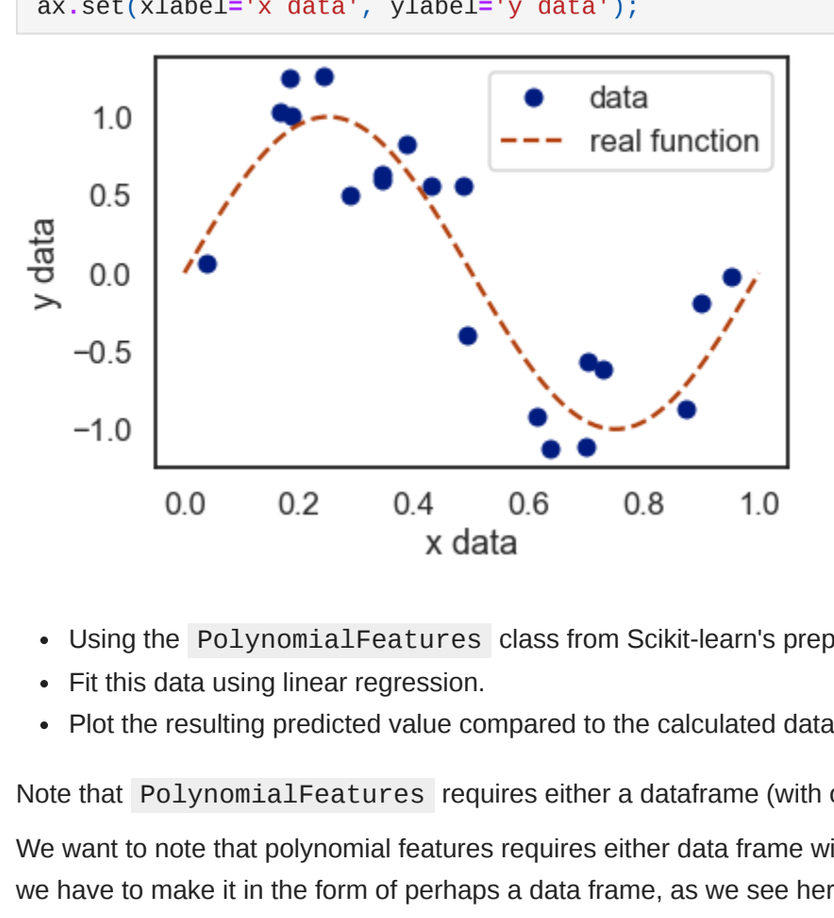
Out[10]:
```
array([0.        , 0.01010101, 0.02020202, 0.03030303, 0.04040404,
       0.05050505, 0.06060606, 0.07070707, 0.08080808, 0.09090909,
       0.1010101 , 0.11111111, 0.12121212, 0.13131313, 0.14141414,
       0.15151515, 0.16161616, 0.17171717, 0.18181818, 0.19191919,
       0.2020202 , 0.21212121, 0.22222222, 0.23232323, 0.24242424,
       0.25252525, 0.26262626, 0.27272727, 0.28282828, 0.29292929,
       0.3030303 , 0.31313131, 0.32323232, 0.33333333, 0.34343434,
       0.35353535, 0.36363636, 0.37373737, 0.38383838, 0.39393939,
       0.4040404 , 0.41414141, 0.42424242, 0.43434343, 0.44444444,
       0.45454545, 0.46464646, 0.47474747, 0.48484848, 0.49494949,
       0.50505051, 0.51515152, 0.52525253, 0.53535354, 0.54545455,
       0.55555556, 0.56565657, 0.57575758, 0.58585859, 0.5959596 ,
       0.60606061, 0.61616162, 0.62626263, 0.63636364, 0.64646465,
       0.65656566, 0.66666667, 0.67676768, 0.68686869, 0.6969697 ,
       0.70707071, 0.71717172, 0.72727273, 0.73737374, 0.74747475,
       0.75757576, 0.76767677, 0.77777778, 0.78787879, 0.7979798 ,
       0.80808081, 0.81818182, 0.82828283, 0.83838384, 0.84848485,
       0.85858586, 0.86868687, 0.87878788, 0.88888889, 0.8989899 ,
       0.90909091, 0.91919192, 0.92929293, 0.93939394, 0.94949495,
       0.95959596, 0.96969697, 0.97979798, 0.98989899, 1.        ])
```

In [12]:
```
Y_real
```

Out[12]:
```
array([ 0.00000000e+00,  6.34239197e-02,  1.26592454e-01,  1.89251244e-01,
        2.51147987e-01,  3.12033446e-01,  3.71662456e-01,  4.29794912e-01,
        4.86196736e-01,  5.40640817e-01,  5.92907929e-01,  6.42787610e-01,
        6.90079011e-01,  7.34591709e-01,  7.76146464e-01,  8.14575952e-01,
        8.49725430e-01,  8.81453363e-01,  9.09631995e-01,  9.34147860e-01,
        9.54902241e-01,  9.71811568e-01,  9.84807753e-01,  9.93838464e-01,
        9.98867339e-01,  9.99874128e-01,  9.96854776e-01,  9.89821442e-01,
        9.78802446e-01,  9.63842159e-01,  9.45000819e-01,  9.22354294e-01,
        8.95993774e-01,  8.66025404e-01,  8.32569855e-01,  7.95761841e-01,
        7.55749574e-01,  7.12694171e-01,  6.66769001e-01,  6.18158986e-01,
        5.67059864e-01,  5.13677392e-01,  4.58226522e-01,  4.00930535e-01,
        3.42020143e-01,  2.81732557e-01,  2.20310533e-01,  1.58001396e-01,
        9.50560433e-02,  3.17279335e-02, -3.17279335e-02, -9.50560433e-02,
       -1.58001396e-01, -2.20310533e-01, -2.81732557e-01, -3.42020143e-01,
       -4.00930535e-01, -4.58226522e-01, -5.13677392e-01, -5.67059864e-01,
       -6.18158986e-01, -6.66769001e-01, -7.12694171e-01, -7.55749574e-01,
       -7.95761841e-01, -8.32569855e-01, -8.66025404e-01, -8.95993774e-01,
       -9.22354294e-01, -9.45000819e-01, -9.63842159e-01, -9.78802446e-01,
       -9.89821442e-01, -9.96854776e-01, -9.99874128e-01, -9.98867339e-01,
       -9.93838464e-01, -9.84807753e-01, -9.71811568e-01, -9.54902241e-01,
       -9.34147860e-01, -9.09631995e-01, -8.81453363e-01, -8.49725430e-01,
       -8.14575952e-01, -7.76146464e-01, -7.34591709e-01, -6.90079011e-01,
       -6.42787610e-01, -5.92907929e-01, -5.40640817e-01, -4.86196736e-01,
       -4.29794912e-01, -3.71662456e-01, -3.12033446e-01, -2.51147987e-01,
       -1.89251244e-01, -1.26592454e-01, -6.34239197e-02, -2.44929360e-16])
```

In [14]:
```
import matplotlib.pyplot as plt
import seaborn as sns

%matplotlib inline
```

In [17]:
```
sns.set_style('white')
sns.set_context('talk')
sns.set_palette('dark')

# Plot of the noisy (sparse)
ax = data.set_index('x')['y'].plot(ls='', marker='o', label='data')
ax.plot(X_real, Y_real, ls='--', marker='', label='real function')

ax.legend()
ax.set(xlabel='x data', ylabel='y data');
```

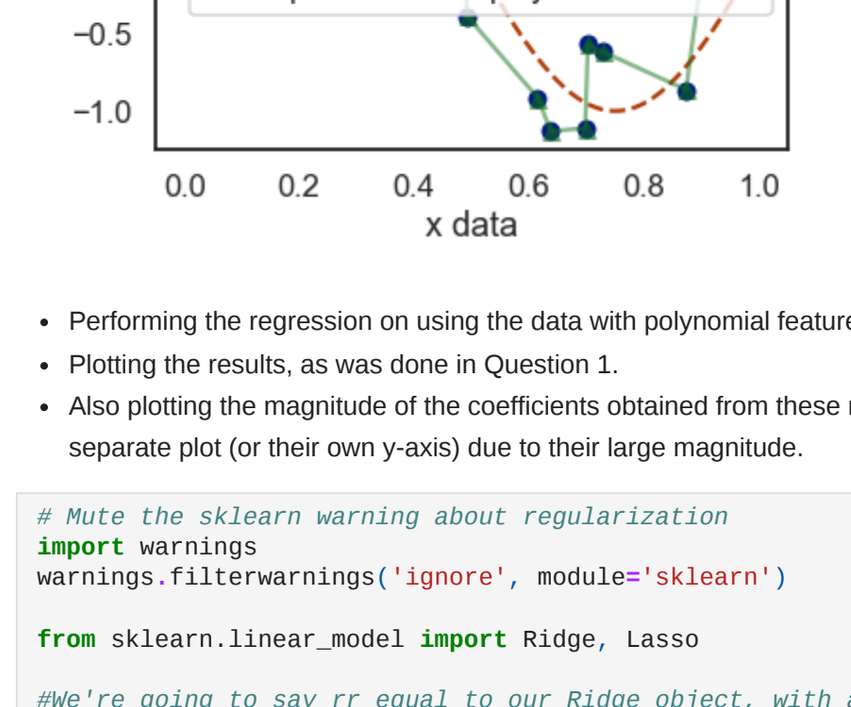


- Using the `PolynomialFeatures` class from Scikit-learn's preprocessing library, create 20th order polynomial features.
- Fit this data using linear regression.
- Plot the resulting predicted value compared to the calculated data.

Note that `PolynomialFeatures` requires either a dataframe (with one column, not a Series) or a 2D array of dimension (`X`, 1), where `X` is the length.

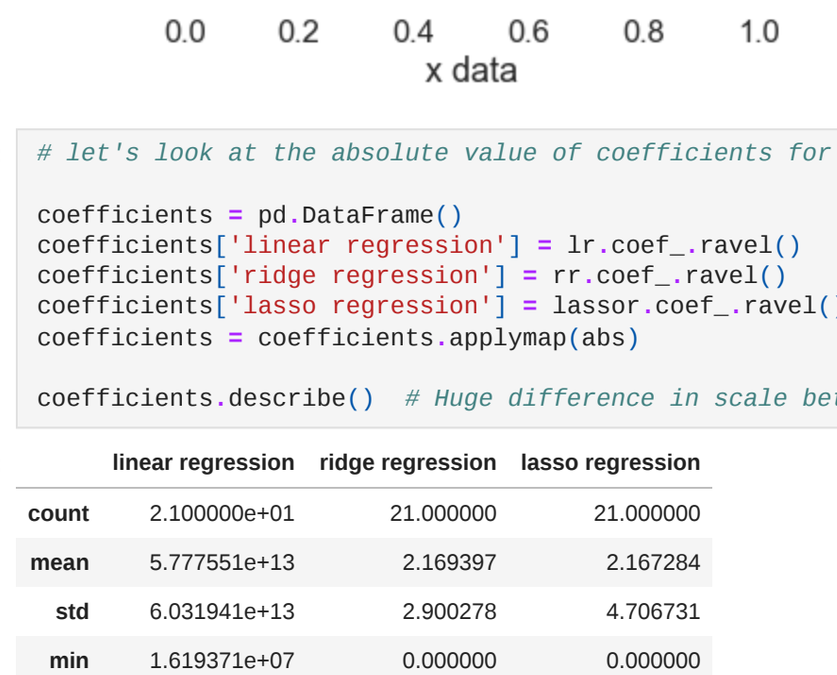
We want to note that polynomial features requires either data frame with one column or a 2D array of dimension (X, 1). we want to ensure that we have a two-dimensional array. we can't just pass in just one column or a series, we have to make it in the form of perhaps a data frame, as we see here, or a two-dimensional array.So we import polynomial features, our linear regression.

In [18]:
```
from sklearn.preprocessing import PolynomialFeatures
from sklearn.linear_model import LinearRegression

# Setup the polynomial features
degree = 20
pf = PolynomialFeatures(degree)
lr = LinearRegression() # use linear regression on that expanded version of X.

# Extract the X- and Y- data from the dataframe
X_data = data[['x']] #  data frame version of that column [[]],if we did a single bracket, we would actually error out here.
Y_data = data['y']

# Create the features and fit the model
X_poly = pf.fit_transform(X_data)
lr = lr.fit(X_poly, Y_data)
Y_pred = lr.predict(X_poly)

# Plot the result
plt.plot(X_data, Y_data, marker='o', ls='', label='data', alpha=1)
plt.plot(X_real, Y_real, ls='--', label='real function')
plt.plot(X_data, Y_pred, marker='^', alpha=.5, label='predictions w/ polynomial features')
plt.legend()
ax = plt.gca()
ax.set(xlabel='x data', ylabel='y data');
```



- Performing the regression on using the data with polynomial features using ridge regression (α=0.001) and lasso regression (α=0.0001).
- Plotting the results, as was done in Question 1.
- Also plotting the magnitude of the coefficients obtained from these regressions, and compare them to those obtained from linear regression in the previous question. The linear regression coefficients will likely need a separate plot (or their own y-axis) due to their large magnitude.

In [21]:
```
# Mute the sklearn warning about regularization
import warnings
warnings.filterwarnings('ignore', module='sklearn')

from sklearn.linear_model import Ridge, Lasso

#We're going to say rr equal to our Ridge object, with an alpha = 0.001.
#We're then going to fit that to our X_poly that we defined before.
#So that's  20 degree polynomial as well as our Y_data.
# The ridge regression model

rr = Ridge(alpha=0.001)
rr = rr.fit(X_poly, Y_data)
Y_pred_rr = rr.predict(X_poly)

# The lasso regression model
lassor = Lasso(alpha=0.0001)
lassor = lassor.fit(X_poly, Y_data)
Y_pred_lr = lassor.predict(X_poly)

# The plot of the predicted values
plt.plot(X_data, Y_data, marker='o', ls='', label='data')
plt.plot(X_real, Y_real, ls='--', label='real function')
plt.plot(X_data, Y_pred, label='linear regression', marker='^', alpha=.5)
plt.plot(X_data, Y_pred_rr, label='ridge regression', marker='^', alpha=.5)
plt.plot(X_data, Y_pred_lr, label='lasso regression', marker='^', alpha=.5)

plt.legend()

ax = plt.gca()
ax.set(xlabel='x data', ylabel='y data');
```



In [23]:
```
# let's look at the absolute value of coefficients for each model

coefficients = pd.DataFrame()
coefficients['linear regression'] = lr.coef_.ravel()
coefficients['ridge regression'] = rr.coef_.ravel()
coefficients['lasso regression'] = lassor.coef_.ravel()
coefficients = coefficients.applymap(abs)

coefficients.describe()  # Huge difference in scale between non-regularized vs regularized regression
```

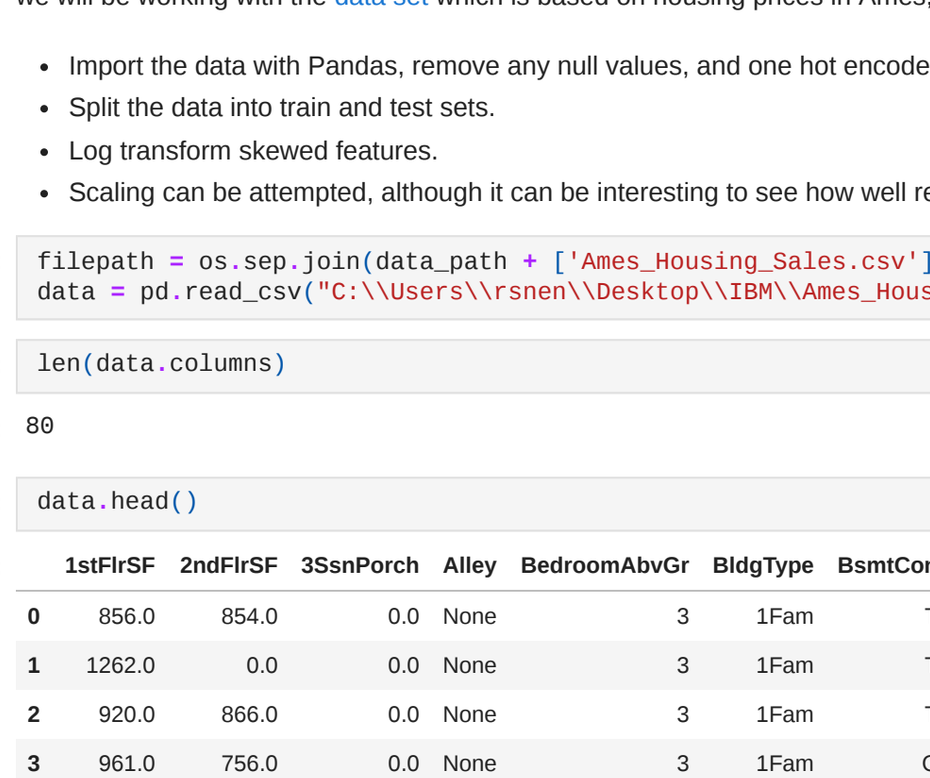
Out[23]:

	linear regression	ridge regression	lasso regression
count	2.100000e+01	21.000000	21.000000
mean	5.777551e+13	2.169397	2.167284
std	6.031941e+13	2.900278	4.706731
min	1.619371e+07	0.000000	0.000000
25%	3.416390e+12	0.467578	0.000000
50%	3.621188e+13	1.017272	0.252181
75%	1.070540e+14	2.883507	1.641353
max	1.655347e+14	12.429635	20.176708

In [25]:
```
(coefficients>0).sum()
```

Out[25]:
```
linear regression    21
ridge regression     20
lasso regression     15
dtype: int64
```

In [37]:
```
colors = sns.color_palette()

# Setup the dual y-axes
ax1 = plt.axes()
ax2 = ax1.twinx()

# Plot the linear regression data
ax1.plot(lr.coef_.ravel(),
         color=colors[0], marker='o', label='linear regression')

# Plot the regularization data sets
ax2.plot(rr.coef_.ravel(),
         color=colors[1], marker='o', label='ridge regression')

ax2.plot(lassor.coef_.ravel(),
         color=colors[2], marker='o', label='lasso regression')

# Customize axes scales
ax1.set_ylim(-2e14, 2e14)
ax2.set_ylim(-25, 25)

# Combine the legends
h1, l1 = ax1.get_legend_handles_labels()
h2, l2 = ax2.get_legend_handles_labels()
ax1.legend(h1+h2, l1+l2)

ax1.set(xlabel='coefficients',ylabel='linear regression')
ax2.set(ylabel='ridge and lasso regression')

ax1.set_xticks(range(len(lr.coef_)));
```



we will be working with the data set which is based on housing prices in Ames, Iowa. There are an extensive number of features To begin:

- Import the data with Pandas, remove any null values, and one hot encode categoricals. Either Scikit-learn's feature encoders or Pandas `get_dummies` method can be used.
- Split the data into train and test sets.
- Log transform skewed features.
- Scaling can be attempted, although it can be interesting to see how well regularization works without scaling features.

In [39]:
```
filepath = os.sep.join(data_path + ['Ames_Housing_Sales.csv'])
data = pd.read_csv("C:\\Users\\rsnen\\Desktop\\IBM\\Ames_Housing_Sales.csv", sep=',')
```

In [40]:
```
len(data.columns)
```

Out[40]:
```
80
```

In [42]:
```
data.head()
```

Out[42]:

	1stFlrSF	2ndFlrSF	3SsnPorch	Alley	BedroomAbvGr	BldgType	BsmtCond	BsmtExposure	BsmtFinSF1	BsmtFinSF2	...	ScreenPorch	Street	TotRmsAbvGrd	TotalBsmtSF	Utilities	WoodDeckSF	YearBuilt	YearRemodAdd
0	856.0	854.0	0.0	None	3	1Fam	TA	No	706.0	0.0	...	0.0	Pave	8	856.0	AllPub	0.0	2003	2003
1	1262.0	0.0	0.0	None	3	1Fam	TA	Gd	978.0	0.0	...	0.0	Pave	6	1262.0	AllPub	298.0	1976	1976
2	920.0	866.0	0.0	None	3	1Fam	TA	Mn	486.0	0.0	...	0.0	Pave	6	920.0	AllPub	0.0	2001	2002
3	961.0	756.0	0.0	None	3	1Fam	Gd	No	216.0	0.0	...	0.0	Pave	7	756.0	AllPub	0.0	1915	1970
4	1145.0	1053.0	0.0	None	4	1Fam	TA	Av	655.0	0.0	...	0.0	Pave	9	1145.0	AllPub	192.0	2000	2000

5 rows × 80 columns

Create a list of categorial data and one-hot encode. Pandas one-hot encoder (`get_dummies`) works well with data that is defined as a categorical.

In [46]:
```
# Do the one hot encoding
data = pd.get_dummies(data, columns=one_hot_encode_cols)
```

In [49]:
```
data = pd.get_dummies(data, drop_first = True)
data.columns
```

Out[49]:
```
Index(['1stFlrSF', '2ndFlrSF', '3SsnPorch', 'BedroomAbvGr', 'BsmtFinSF1',
       'BsmtFinSF2', 'BsmtFullBath', 'BsmtHalfBath', 'BsmtUnfSF',
       'EnclosedPorch',
       ...
       'SaleType_ConLD', 'SaleType_ConLI', 'SaleType_ConLw', 'SaleType_New',
       'SaleType_Oth', 'SaleType_WD', 'Street_Grvl', 'Street_Pave',
       'Utilities_AllPub', 'Utilities_NoSeWa'],
      dtype='object', length=295)
```

Next, split the data in train and test data sets.

In [51]:
```
from sklearn.model_selection import train_test_split

train, test = train_test_split(data, test_size=0.3, random_state=42)
```

There are a number of columns that have skewed features--a log transformation can be applied to them. Note that this includes the `SalePrice`, our predictor. However, let's keep that one as is.

In [54]:
```
# Create a list of float colums to check for skewing
mask = data.dtypes == np.float
float_cols = data.columns[mask]
```

We're then going to take that, filter down here to only those that are above 0.75, we're then going to sort those values from largest to smallest, we're then going to put it into a DataFrame. So from Pandas series, we have the method to translate it into a Pandas DataFrame. Since it's now a new Pandas DataFrame, we're then going to have a default of a column called zero. We're just going to rename that skew.

In [57]:
```
skew_limit = 0.75
skew_vals = train[float_cols].skew()

skew_cols = (skew_vals
             .sort_values(ascending=False)
             .to_frame()
             .rename(columns={0:'Skew'})
             .query('abs(Skew) > {0}'.format(skew_limit)))

skew_cols
```

Out[57]:

	Skew
MiscVal	26.915364
PoolArea	15.777668
LotArea	11.501694
LowQualFinSF	11.210638
3SsnPorch	10.150612
ScreenPorch	4.599803
BsmtFinSF2	4.466378
EnclosedPorch	3.218303
LotFrontage	3.138032
MasVnrArea	2.492814
OpenPorchSF	2.295489
SalePrice	2.106910
BsmtFinSF1	2.010766
TotalBsmtSF	1.979164
1stFlrSF	1.539692
GrLivArea	1.455564
WoodDeckSF	1.334388
BsmtUnfSF	0.900308
GarageArea	0.838422
2ndFlrSF	0.773655

Transform all the columns where the skew is greater than 0.75, excluding "SalePrice".

In [60]:
```
# OPTIONAL: Let's look at what happens to one of these features, when we apply np.log1p visually.

field = "BsmtFinSF1"
fig, (ax_before, ax_after) = plt.subplots(1, 2, figsize=(10, 5))
train[field].hist(ax=ax_before)
train[field].apply(np.log1p).hist(ax=ax_after)
ax_before.set(title='before np.log1p', ylabel='frequency', xlabel='value')
ax_after.set(title='after np.log1p', ylabel='frequency', xlabel='value')
fig.suptitle('Field "{}"'.format(field));
# a little bit better
```

In [62]:
```
# Mute the setting wtih a copy warnings
pd.options.mode.chained_assignment = None

for col in skew_cols.index.tolist():
    if col == "SalePrice":
        continue
    train[col] = np.log1p(train[col])
    test[col]  = test[col].apply(np.log1p)  # same thing
```

Separate features from predictor.

In [65]:
```
feature_cols = [x for x in train.columns if x != 'SalePrice']
X_train = train[feature_cols]
y_train = train['SalePrice']

X_test  = test[feature_cols]
y_test  = test['SalePrice']
```

- Writing a function **`rmse`** that takes in truth and prediction values and returns the root-mean-squared error. Use sklearn's `mean_squared_error`.

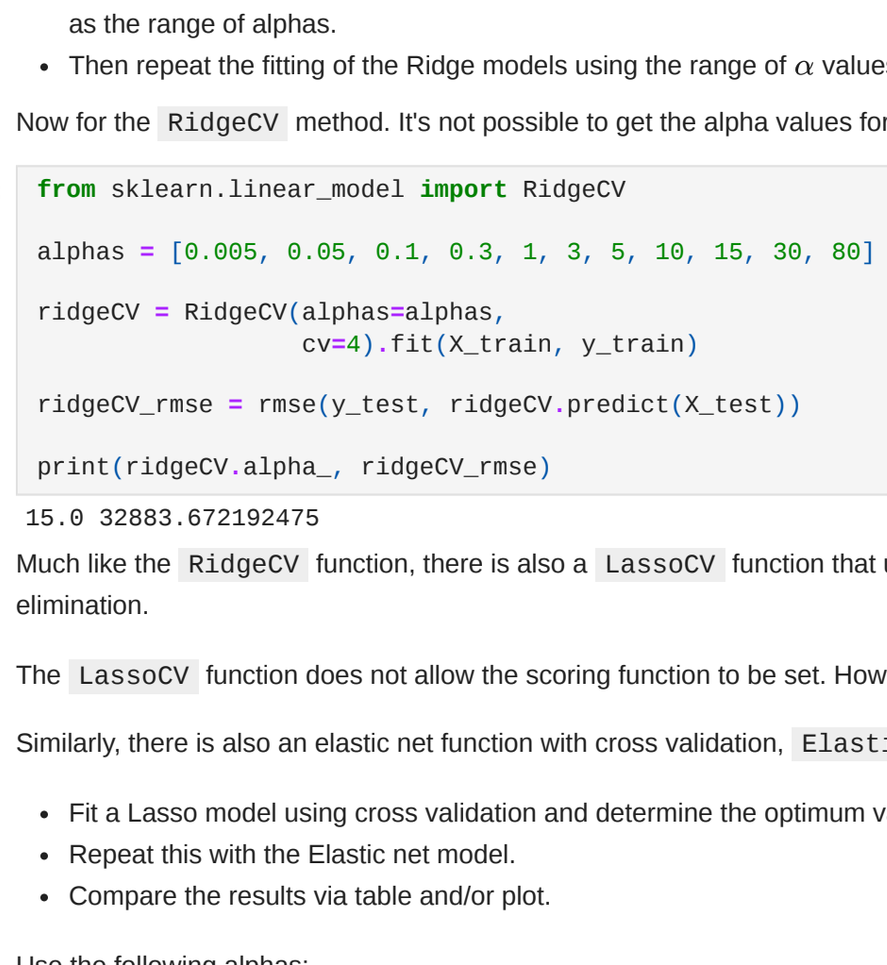
In [68]:
```
from sklearn.metrics import mean_squared_error


def rmse(ytrue, ypredicted):
    return np.sqrt(mean_squared_error(ytrue, ypredicted))
```

- Fit a basic linear regression model
- print the root-mean-squared error for this model
- plot the predicted vs actual sale price based on the model.

In [72]:
```
from sklearn.linear_model import LinearRegression

linearRegression = LinearRegression().fit(X_train, y_train)

linearRegression_rmse = rmse(y_test, linearRegression.predict(X_test))

print(linearRegression_rmse)
```

```
633477.1807710954
```

In [75]:
```
f = plt.figure(figsize=(6,6))
ax = plt.axes()

ax.plot(y_test, linearRegression.predict(X_test),
         marker='o', ls='', ms=3.0)

lim = (0, y_test.max())

ax.set(xlabel='Actual Price',
       ylabel='Predicted Price',
       xlim=lim,
       ylim=lim,
       title='Linear Regression Results');
```



Ridge regression uses L2 normalization to reduce the magnitude of the coefficients. This can be helpful in situations where there is high variance. The regularization functions in Scikit-learn each contain versions that have cross-validation built in.

- Fit a regular (non-cross validated) Ridge model to a range of α values and plot the RMSE using the cross validated error function you created above.
- Use

$$[0.005, 0.05, 0.1, 0.3, 1, 3, 5, 10, 15, 30, 80]$$

  as the range of alphas.
- Then repeat the fitting of the Ridge models using the range of α values from the prior section. Compare the results.

Now for the `RidgeCV` method. It's not possible to get the alpha values for the models that weren't selected, unfortunately. The resulting error values and α values are very similar to those obtained above.

In [77]:
```
from sklearn.linear_model import RidgeCV

alphas = [0.005, 0.05, 0.1, 0.3, 1, 3, 5, 10, 15, 30, 80]

ridgeCV = RidgeCV(alphas=alphas,
                  cv=4).fit(X_train, y_train)

ridgeCV_rmse = rmse(y_test, ridgeCV.predict(X_test))

print(ridgeCV.alpha_, ridgeCV_rmse)
```

```
15.0 32883.672192475
```

Much like the `RidgeCV` function, there is also a `LassoCV` function that uses an L1 regularization function and cross-validation. L1 regularization will selectively shrink some coefficients, effectively performing feature elimination.

The `LassoCV` function does not allow the scoring function to be set. However, the custom error function (`rmse`) created above can be used to evaluate the error on the final model.

Similarly, there is also an elastic net function with cross validation, `ElasticNetCV`, which is a combination of L2 and L1 regularization.

- Fit a Lasso model using cross validation and determine the optimum value for α and the RMSE using the function created above. Note that the magnitude of α may be different from the Ridge model.
- Repeat this with the Elastic net model.
- Compare the results via table and/or plot.

Use the following alphas:
`[1e-5, 5e-5, 0.0001, 0.0005]`

In [79]:
```
from sklearn.linear_model import LassoCV

alphas2 = np.array([1e-5, 5e-5, 0.0001, 0.0005])

lassoCV = LassoCV(alphas=alphas2,
                  max_iter=5e4,
                  cv=3).fit(X_train, y_train)

lassoCV_rmse = rmse(y_test, lassoCV.predict(X_test))

print(lassoCV.alpha_, lassoCV_rmse)  # Lasso is slower
```

```
0.0005 43180.91016592929
```

We can determine how many of these features remain non-zero.

In [82]:
```
print('Of {} coefficients, {} are non-zero with Lasso.'.format(len(lassoCV.coef_),
                                                              len(lassoCV.coef_.nonzero()[0])))
```

```
Of 294 coefficients, 275 are non-zero with Lasso.
```

Now lets try the elastic net, with the same alphas as in Lasso, and l1_ratios between 0.1 and 0.9

In [83]:
```
from sklearn.linear_model import ElasticNetCV

l1_ratios = np.linspace(0.1, 0.9, 9)

elasticNetCV = ElasticNetCV(alphas=alphas2,
                            l1_ratio=l1_ratios,
                            max_iter=1e4).fit(X_train, y_train)
elasticNetCV_rmse = rmse(y_test, elasticNetCV.predict(X_test))

print(elasticNetCV.alpha_, elasticNetCV.l1_ratio_, elasticNetCV_rmse)
```

```
0.0005 0.1 35206.951603170026
```

Comparing the RMSE calculation from all models is easiest in a table.

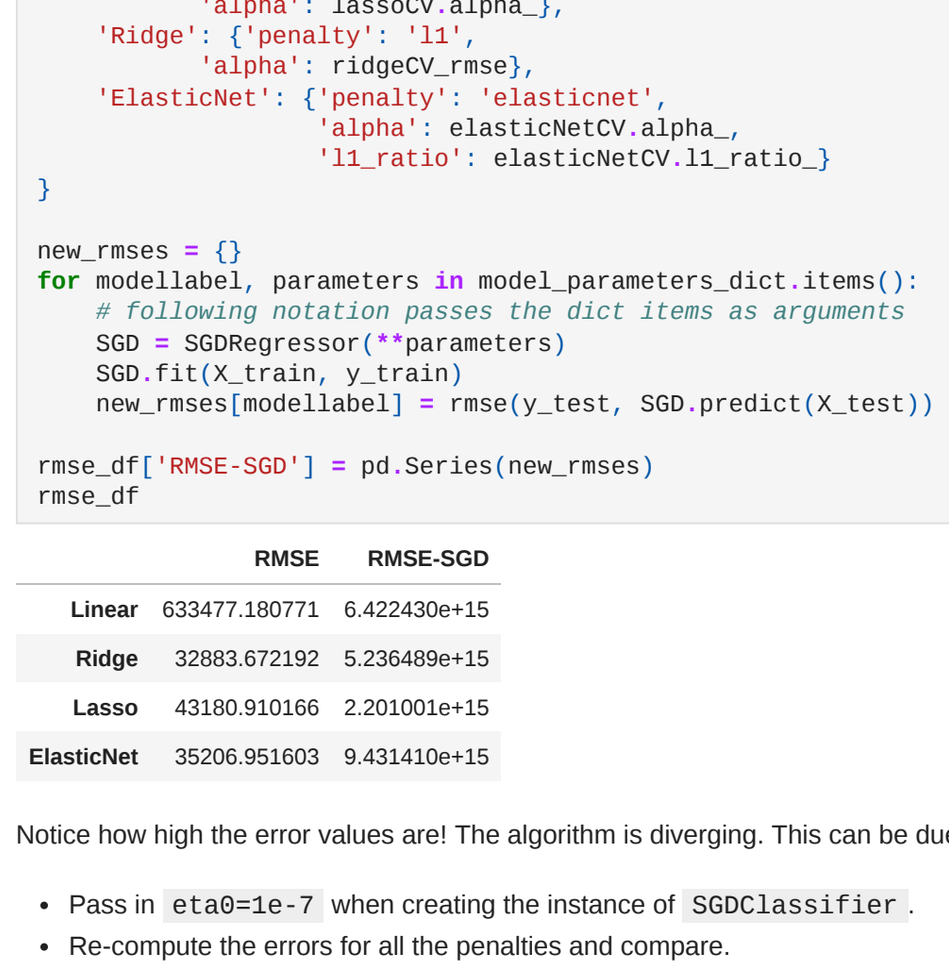
In [85]:
```
rmse_vals = [linearRegression_rmse, ridgeCV_rmse, lassoCV_rmse, elasticNetCV_rmse]

labels = ['Linear', 'Ridge', 'Lasso', 'ElasticNet']

rmse_df = pd.Series(rmse_vals, index=labels).to_frame()
rmse_df.rename(columns={0: 'RMSE'}, inplace=1)
rmse_df
```

Out[85]:

	RMSE
Linear	633477.180771
Ridge	32883.672192
Lasso	43180.910166
ElasticNet	35206.951603

We can also make a plot of actual vs predicted housing prices as before.

In [87]:
```
f = plt.figure(figsize=(6,6))
ax = plt.axes()

labels = ['Ridge', 'Lasso', 'ElasticNet']

models = [ridgeCV, lassoCV, elasticNetCV]

for mod, lab in zip(models, labels):
    ax.plot(y_test, mod.predict(X_test),
             marker='o', ls='', ms=3.0, label=lab)


leg = plt.legend(frameon=True)
leg.get_frame().set_edgecolor('black')
leg.get_frame().set_linewidth(1.0)

ax.set(xlabel='Actual Price',
       ylabel='Predicted Price',
       title='Linear Regression Results');
```



Let's explore Stochastic gradient descent in this exercise.
Recall that Linear models in general are sensitive to scaling. However, SGD is *very* sensitive to scaling.
Moreover, a high value of learning rate can cause the algorithm to diverge, whereas a too low value may take too long to converge.

- Fit a stochastic gradient descent model without a regularization penalty (the relevant parameter is `penalty`).
- Now fit stochastic gradient descent models with each of the three penalties (L2, L1, Elastic Net) using the parameter values determined by cross validation above.
- Do not scale the data before fitting the model.
- Compare the results to those obtained without using stochastic gradient descent.

In [90]:
```
# Import SGDRegressor and prepare the parameters

from sklearn.linear_model import SGDRegressor

model_parameters_dict = {
    'Linear': {'penalty': 'none'},
    'Lasso': {'penalty': 'l2',
           'alpha': lassoCV.alpha_},
    'Ridge': {'penalty': 'l1',
           'alpha': ridgeCV_rmse},
    'ElasticNet': {'penalty': 'elasticnet',
                   'alpha': elasticNetCV.alpha_,
                   'l1_ratio': elasticNetCV.l1_ratio_}
}

new_rmses = {}
for modellabel, parameters in model_parameters_dict.items():
    # following notation passes the dict items as arguments
    SGD = SGDRegressor(**parameters)
    SGD.fit(X_train, y_train)
    new_rmses[modellabel] = rmse(y_test, SGD.predict(X_test))

rmse_df['RMSE-SGD'] = pd.Series(new_rmses)
rmse_df
```

Out[90]:

	RMSE	RMSE-SGD
Linear	633477.180771	6.422430e+15
Ridge	32883.672192	5.236489e+15
Lasso	43180.910166	2.201001e+15
ElasticNet	35206.951603	9.431410e+15

Notice how high the error values are! The algorithm is diverging. This can be due to scaling and/or learning rate being too high. Let's adjust the learning rate and see what happens.

- Pass in `eta0=1e-7` when creating the instance of `SGDClassifier`.
- Re-compute the errors for all the penalties and compare.

In [92]:
```
# Import SGDRegressor and prepare the parameters

from sklearn.linear_model import SGDRegressor

model_parameters_dict = {
    'Linear': {'penalty': 'none'},
    'Lasso': {'penalty': 'l2',
           'alpha': lassoCV.alpha_},
    'Ridge': {'penalty': 'l1',
           'alpha': ridgeCV_rmse},
    'ElasticNet': {'penalty': 'elasticnet',
                   'alpha': elasticNetCV.alpha_,
                   'l1_ratio': elasticNetCV.l1_ratio_}
}

new_rmses = {}
for modellabel, parameters in model_parameters_dict.items():
    # following notation passes the dict items as arguments
    SGD = SGDRegressor(eta0=1e-7, **parameters)
    SGD.fit(X_train, y_train)
    new_rmses[modellabel] = rmse(y_test, SGD.predict(X_test))

rmse_df['RMSE-SGD-learningrate'] = pd.Series(new_rmses)
rmse_df
```

Out[92]:

	RMSE	RMSE-SGD	RMSE-SGD-learningrate
Linear	633477.180771	6.422430e+15	72821.103928
Ridge	32883.672192	5.236489e+15	75389.918918
Lasso	43180.910166	2.201001e+15	75155.270525
ElasticNet	35206.951603	9.431410e+15	74218.759038

Now let's scale our training data and try again.

- Fit a `MinMaxScaler` to `X_train` create a variable `X_train_scaled`.
- Using the scaler, transform `X_test` and create a variable `X_test_scaled`.
- Apply the same versions of SGD to them and compare the results. Don't pass in a eta0 this time.

In [94]:
```
from sklearn.preprocessing import MinMaxScaler

scaler = MinMaxScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)

new_rmses = {}
for modellabel, parameters in model_parameters_dict.items():
    # following notation passes the dict items as arguments
    SGD = SGDRegressor(**parameters)
    SGD.fit(X_train_scaled, y_train)
    new_rmses[modellabel] = rmse(y_test, SGD.predict(X_test_scaled))

rmse_df['RMSE-SGD-scaled'] = pd.Series(new_rmses)
rmse_df
```

Out[94]:

	RMSE	RMSE-SGD	RMSE-SGD-learningrate	RMSE-SGD-scaled
Linear	633477.180771	6.422430e+15	72821.103928	33137.659840
Ridge	32883.672192	5.236489e+15	75389.918918	77755.536476
Lasso	43180.910166	2.201001e+15	75155.270525	32983.389373
ElasticNet	35206.951603	9.431410e+15	74218.759038	33461.393577

In [96]:
```
from sklearn.preprocessing import MinMaxScaler

scaler = MinMaxScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)

new_rmses = {}
for modellabel, parameters in model_parameters_dict.items():
    # following notation passes the dict items as arguments
    SGD = SGDRegressor(**parameters)
    SGD.fit(X_train_scaled, y_train)
    new_rmses[modellabel] = rmse(y_test, SGD.predict(X_test_scaled))

rmse_df['RMSE-SGD-scaled'] = pd.Series(new_rmses)
rmse_df
```

Out[96]:

	RMSE	RMSE-SGD	RMSE-SGD-learningrate	RMSE-SGD-scaled
Linear	633477.180771	6.422430e+15	72821.103928	33399.992821
Ridge	32883.672192	5.236489e+15	75389.918918	77772.380255
Lasso	43180.910166	2.201001e+15	75155.270525	33237.088478
ElasticNet	35206.951603	9.431410e+15	74218.759038	33437.882444