RegularizationAndGradientDescent

April 4, 2023

1 Polynomial Features and Regularization

Regularization and Gradient Descent

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`.

Task1

Import 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.

[1]:
```
# Surpress warnings from using older version of sklearn:
def warn(*args, **kwargs):
    pass
import warnings
warnings.warn = warn
```

[5]:
```
import pandas as pd
import numpy as np


data = pd.read_csv("https://cf-courses-data.s3.us.cloud-object-storage.
 ↪appdomain.cloud/IBM-ML240EN-SkillsNetwork/labs/data/X_Y_Sinusoid_Data.csv")
data.head()
```

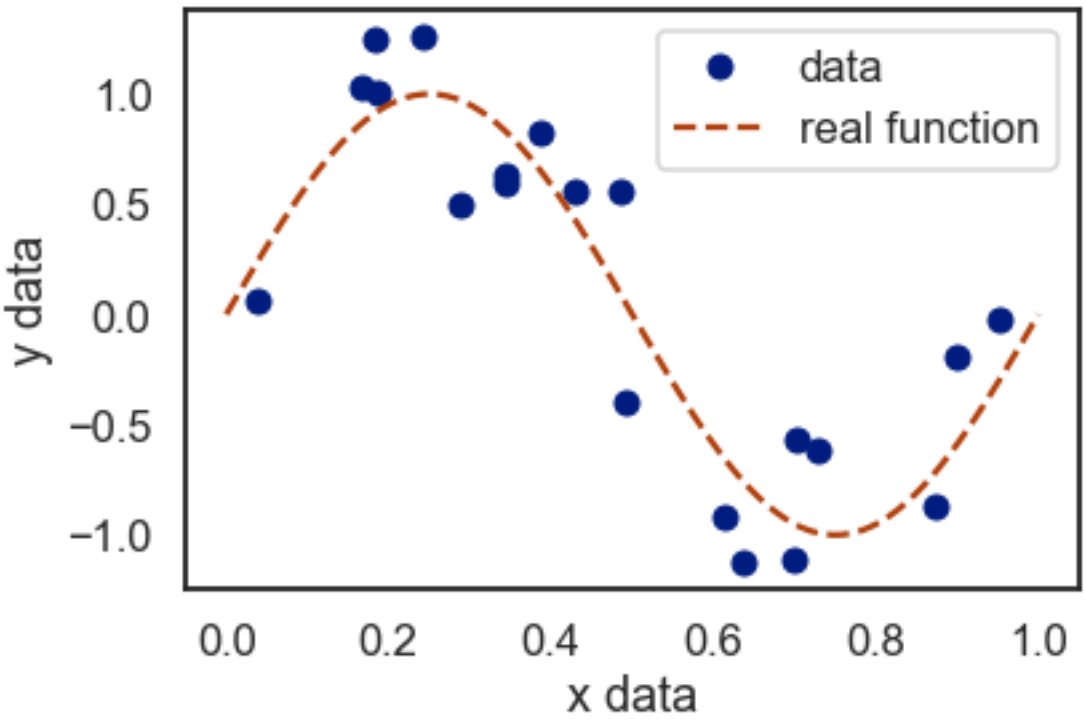
[5]:

	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

[6]:
```
X_real = np.linspace(0, 1.0, 100)
Y_real = np.sin(2 * np.pi * X_real)
```

[3]:
```
import matplotlib.pyplot as plt
import seaborn as sns

%matplotlib inline
```

[4]:
```
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');
```



Task2

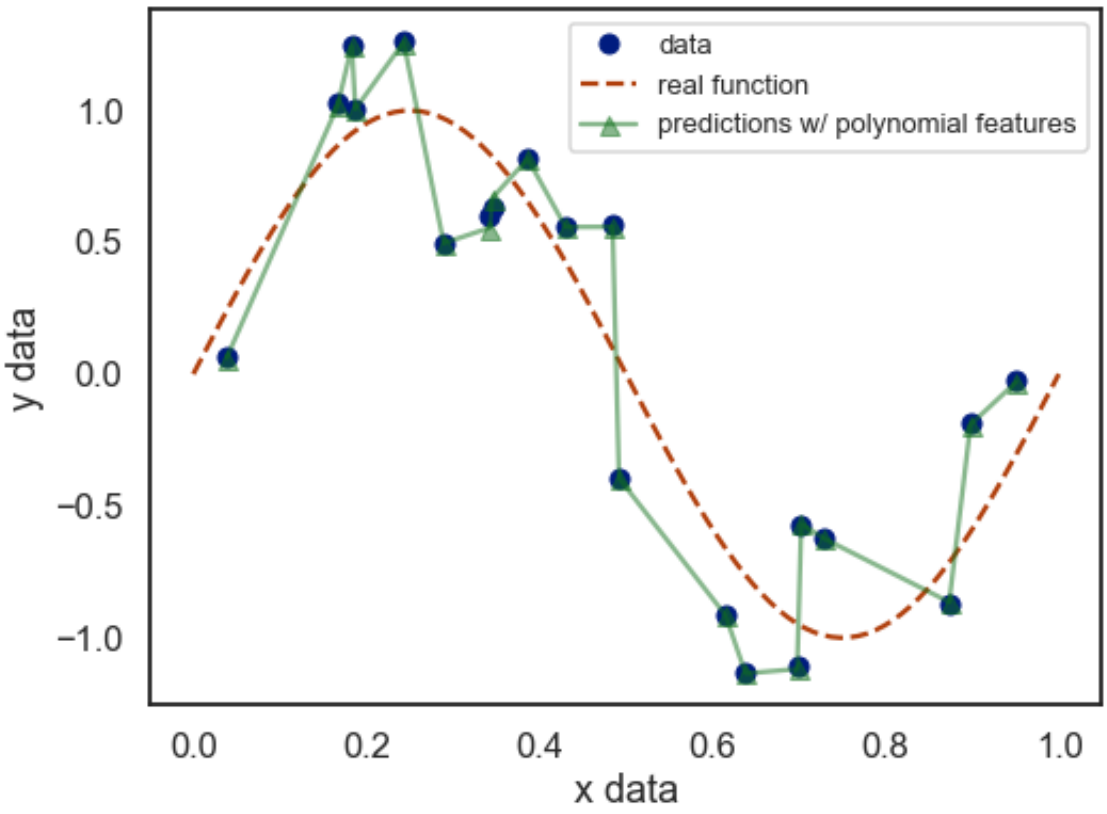
Using the PolynomialFeatures class from Scikit-learn's preprocessing library, create 20th order polynomial features.

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.

[15]:
```
from sklearn.preprocessing import PolynomialFeatures
from sklearn.linear_model import LinearRegression

# Setup the polynomial features
degree = 20
pf = PolynomialFeatures(degree)
lr = LinearRegression()

# Extract the X- and Y- data from the dataframe
X_data = data[['x']]
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.figure(figsize=(8, 6))
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(loc='upper right', bbox_to_anchor=(1.0, 1.0), fontsize=13)

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



Task3

Perform the regression on using the data with polynomial features using ridge regression (α=0.001) and lasso regression (α=0.0001).

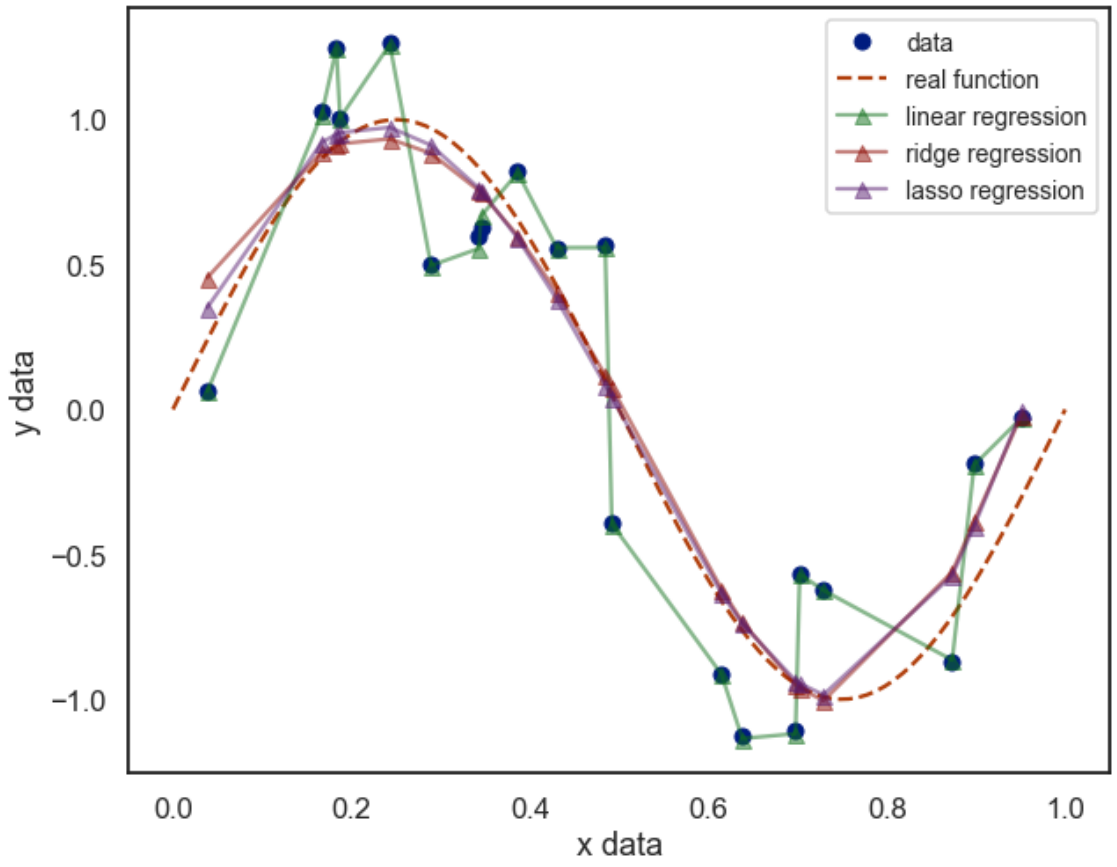
Plot the results, as was done in Task1.

Also plot 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.

What does the comparatively large magnitude of the data tell you about the role of regularization?

[18]:
```
# Mute the sklearn warning about regularization
import warnings
warnings.filterwarnings('ignore', module='sklearn')

from sklearn.linear_model import Ridge, Lasso

# 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.figure(figsize=(10, 8))

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(loc='upper right', bbox_to_anchor=(1.0, 1.0), fontsize=14)

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



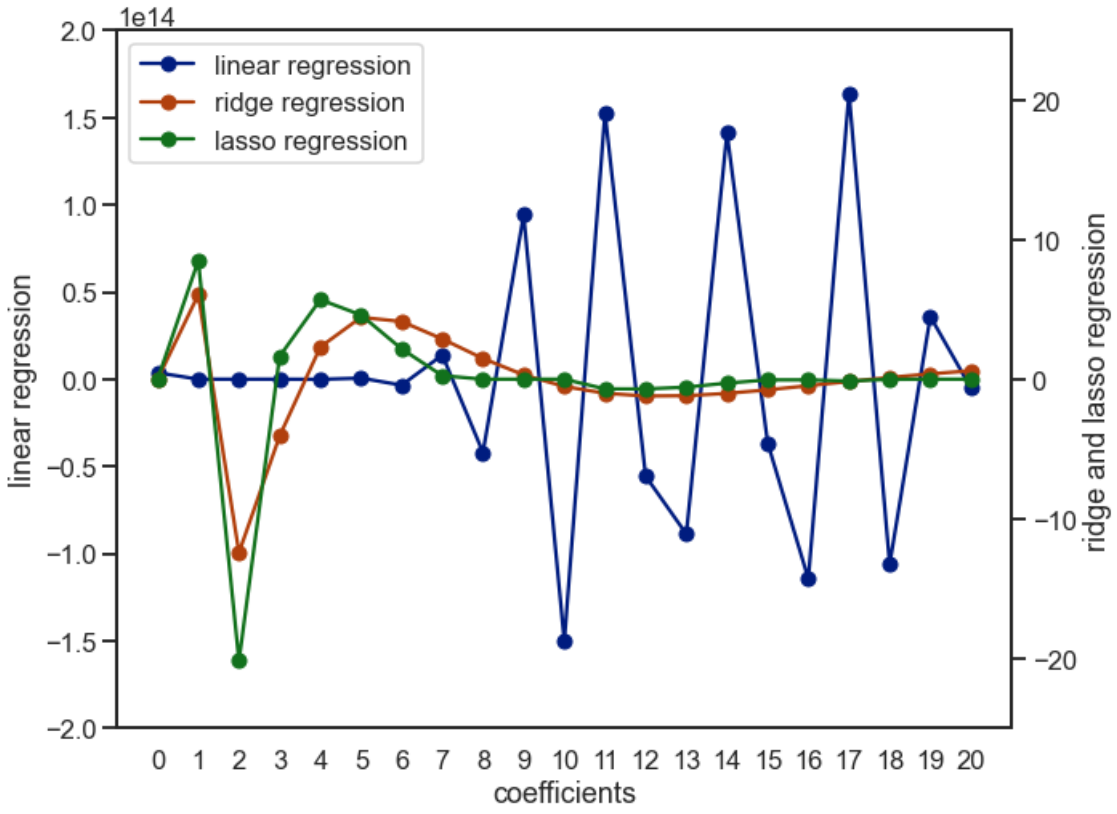
[19]:
```
# let's look at the absolute value of coefficients for each model

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
```

[19]:

	linear regression	ridge regression	lasso regression
count	2.100000e+01	21.000000	21.000000
mean	5.754304e+13	2.169397	2.167284
std	5.999233e+13	2.900278	4.706731
min	1.611590e+07	0.000000	0.000000
25%	3.403676e+12	0.467578	0.000000
50%	3.649017e+13	1.017272	0.252181
75%	1.061917e+14	2.883507	1.641353
max	1.639660e+14	12.429635	20.176708

[22]:
```
import matplotlib.pyplot as plt
import seaborn as sns

colors = sns.color_palette()

# Setup the dual y-axes
fig, ax1 = plt.subplots(figsize=(10, 8))
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_)));
```



Task4

For the remaining questions, we will be working with the data set, which is based on housing prices in Ames, Iowa. There are an extensive number of features–see the exercises from week three for a discussion of these 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.

[23]:
```
data = pd.read_csv("https://cf-courses-data.s3.us.cloud-object-storage.
↪appdomain.cloud/IBM-ML240EN-SkillsNetwork/labs/data/Ames_Housing_Sales.csv")
data.head(10)
```

[23]:

	1stFlrSF	2ndFlrSF	3SsnPorch	Alley	BedroomAbvGr	BldgType	BsmtCond	\
0	856.0	854.0	0.0	None	3	1Fam	TA	
1	1262.0	0.0	0.0	None	3	1Fam	TA	
2	920.0	866.0	0.0	None	3	1Fam	TA	
3	961.0	756.0	0.0	None	3	1Fam	Gd	
4	1145.0	1053.0	0.0	None	4	1Fam	TA	
5	796.0	566.0	320.0	None	1	1Fam	TA	
6	1694.0	0.0	0.0	None	3	1Fam	TA	
7	1107.0	983.0	0.0	None	3	1Fam	TA	
8	1022.0	752.0	0.0	None	2	1Fam	None	
9	1077.0	0.0	0.0	None	2	2fmCon	TA	

	BsmtExposure	BsmtFinSF1	BsmtFinSF2	…	ScreenPorch	Street	TotRmsAbvGrd	\
0	No	706.0	0.0	…	0.0	Pave	8	
1	Gd	978.0	0.0	…	0.0	Pave	6	
2	Mn	486.0	0.0	…	0.0	Pave	6	
3	No	216.0	0.0	…	0.0	Pave	7	
4	Av	655.0	0.0	…	0.0	Pave	9	
5	No	732.0	0.0	…	0.0	Pave	5	
6	Av	1369.0	0.0	…	0.0	Pave	7	
7	Mn	859.0	32.0	…	0.0	Pave	7	
8	None	0.0	0.0	…	0.0	Pave	8	
9	No	851.0	0.0	…	0.0	Pave	5	

	TotalBsmtSF	Utilities	WoodDeckSF	YearBuilt	YearRemodAdd	YrSold	SalePrice
0	856.0	AllPub	0.0	2003	2003	2008	208500.0
1	1262.0	AllPub	298.0	1976	1976	2007	181500.0
2	920.0	AllPub	0.0	2001	2002	2008	223500.0
3	756.0	AllPub	0.0	1915	1970	2006	140000.0
4	1145.0	AllPub	192.0	2000	2000	2008	250000.0
5	796.0	AllPub	40.0	1993	1995	2009	143000.0
6	1686.0	AllPub	255.0	2004	2005	2007	307000.0
7	1107.0	AllPub	235.0	1973	1973	2009	200000.0
8	952.0	AllPub	90.0	1931	1950	2008	129900.0
9	991.0	AllPub	0.0	1939	1950	2008	118000.0

[10 rows x 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.

[25]:
```
# Get a Pd.Series consisting of all the string categoricals
one_hot_encode_cols = data.dtypes[data.dtypes == object]  # filtering by string␣
 ↪categoricals
one_hot_encode_cols = one_hot_encode_cols.index.tolist()  # list of categorical␣
 ↪fields

# Here we see another way of one-hot-encoding:
# Encode these columns as categoricals so one hot encoding works on split data␣
 ↪(if desired)
for col in one_hot_encode_cols:
    data[col] = pd.Categorical(data[col])

# Do the one hot encoding
data = pd.get_dummies(data, columns=one_hot_encode_cols)
```

[26]:
```
# split the data in train and test data sets.
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.

[27]:
```
# Create a list of float colums to check for skewing
mask = data.dtypes == float
float_cols = data.columns[mask]
```

[28]:
```
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
```

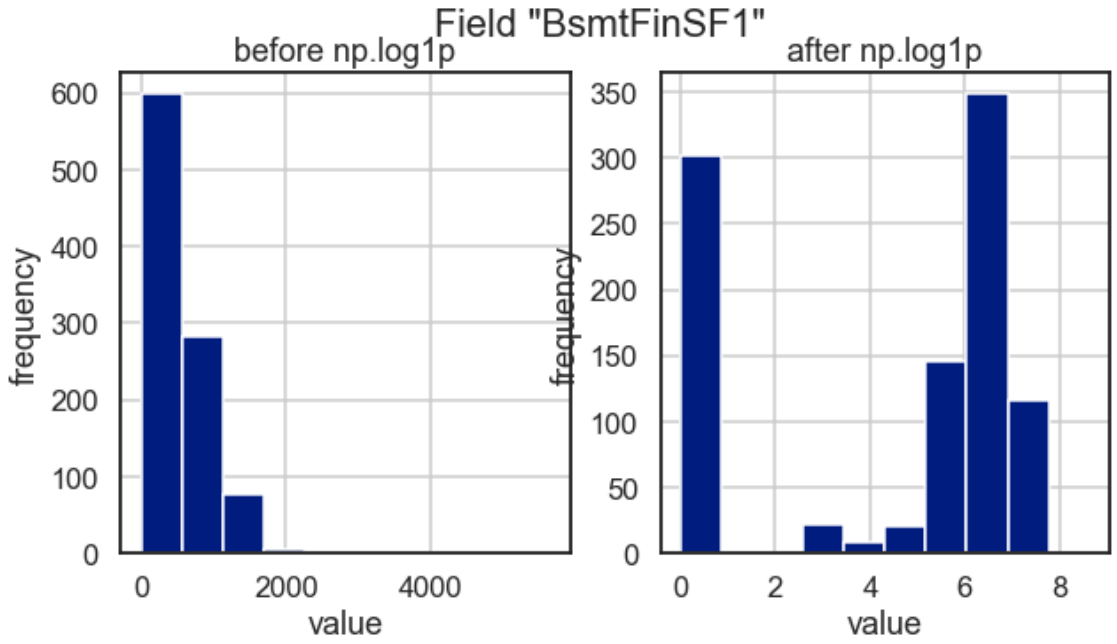
[28]:

	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".

[29]:
```
# 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
```



[30]:
```
# 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
```

[31]:
```
# Separate features from predictor.
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']
```

Task5

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

[32]:
```
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

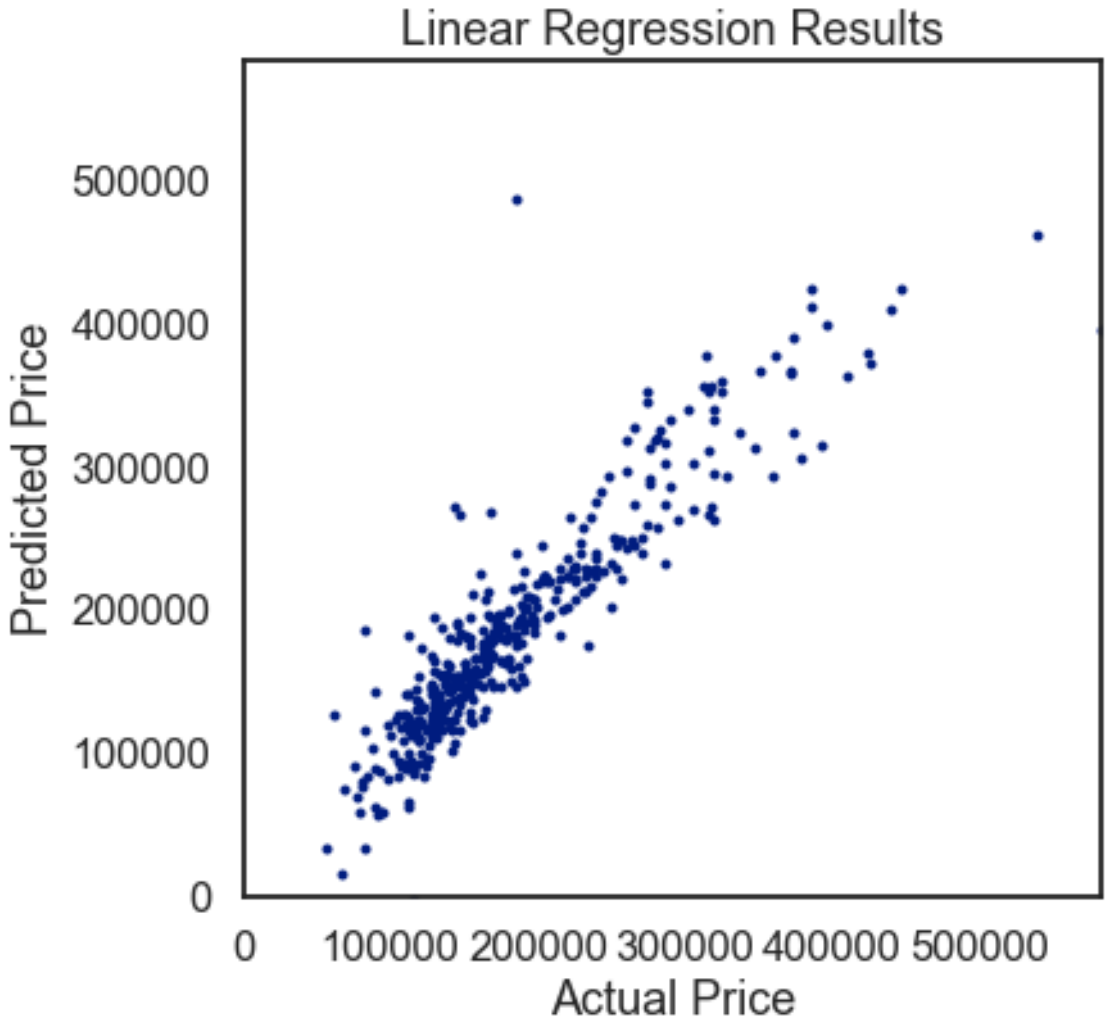
plot the predicted vs actual sale price based on the model.

[33]:
```
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)
```

```
306369.6834231985
```

[34]:
```
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');
```



Task6

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.

[37]:
```
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.alpha_,"\nridgeCV rmse: ", ridgeCV_rmse)
```

```
ridgeCV alpha:  15.0
ridgeCV rmse:  32169.17620567245
```

Task7

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]`

[38]:
```
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 39257.393991449186
```

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

[39]:
```
print('Of {} coefficients, {} are non-zero with Lasso.'.format(len(lassoCV.
 ↪coef_),
                                                               len(lassoCV.
 ↪coef_.nonzero()[0])))
```

```
Of 294 coefficients, 273 are non-zero with Lasso.
```

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

[41]:
```
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.alpha_, "\nelasticNetCV l1_ratio: ",␣
 ↪elasticNetCV.l1_ratio_, "\nelasticNetCV rmse: ", elasticNetCV_rmse)
```

```
elasticNetCV alpha:  0.0005
elasticNetCV l1_ratio:  0.1
elasticNetCV rmse:  35001.23429607459
```

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

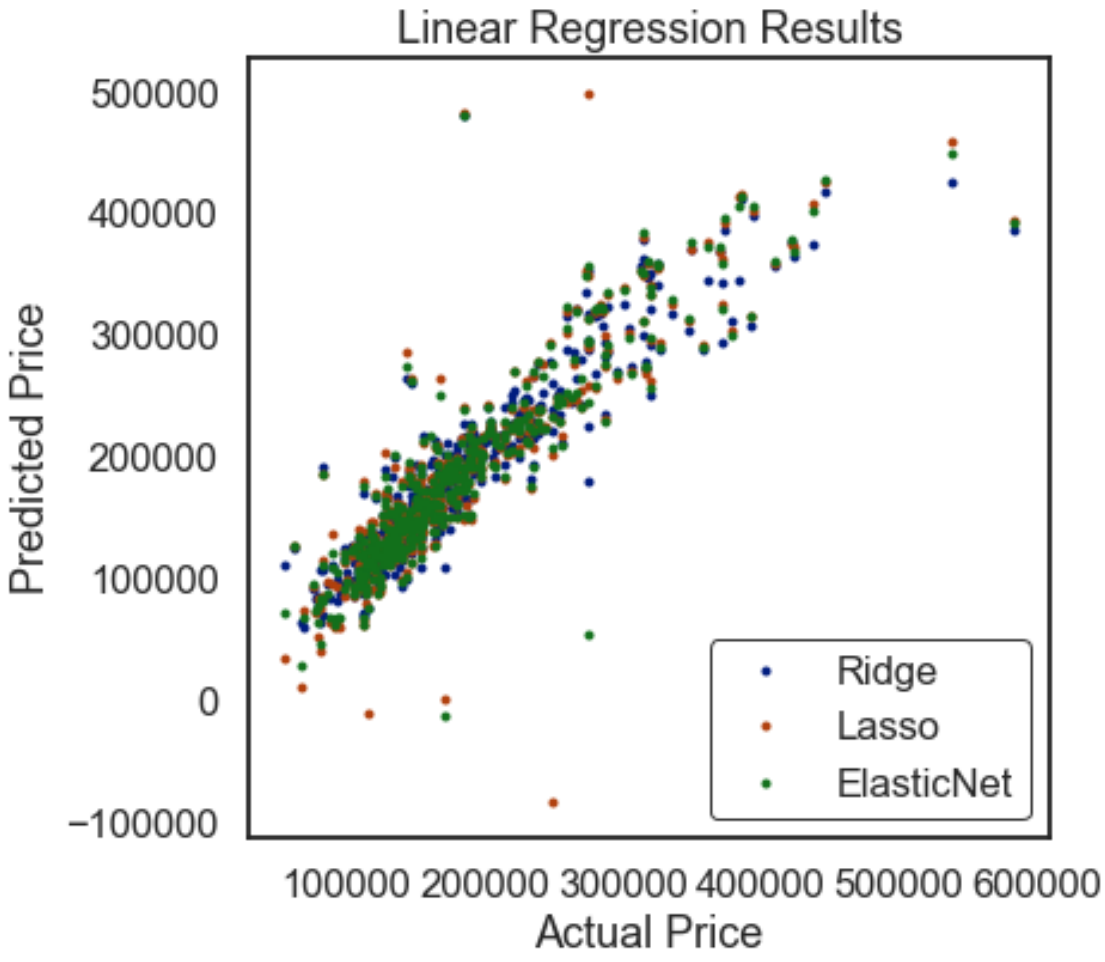
[42]:
```
rmse_vals = [linearRegression_rmse, ridgeCV_rmse, lassoCV_rmse,␣
 ↪elasticNetCV_rmse]

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
```

[42]:

	RMSE
Linear	306369.683423
Ridge	32169.176206
Lasso	39257.393991
ElasticNet	35001.234296

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

[43]:
```
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');
```



Task8

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.

[44]:
```
# 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
```

[44]:

	RMSE	RMSE-SGD
Linear	306369.683423	1.074298e+16
Ridge	32169.176206	6.422933e+15
Lasso	39257.393991	4.864477e+15

```
ElasticNet   35001.234296  8.694994e+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.

[45]:
```
# 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
```

[45]:
```
                     RMSE      RMSE-SGD  RMSE-SGD-learningrate
Linear     306369.683423  1.074298e+16           78006.308526
Ridge       32169.176206  6.422933e+15           76905.859017
Lasso       39257.393991  4.864477e+15           81633.200993
ElasticNet  35001.234296  8.694994e+15           72000.166118
```

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.

[46]:
```
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
```

[46]:

	RMSE	RMSE-SGD	RMSE-SGD-learningrate	RMSE-SGD-scaled
Linear	306369.683423	1.074298e+16	78006.308526	32785.211341
Ridge	32169.176206	6.422933e+15	76905.859017	77776.160547
Lasso	39257.393991	4.864477e+15	81633.200993	32791.985461
ElasticNet	35001.234296	8.694994e+15	72000.166118	32881.304540