S-109A Introduction to Data Science

Homework 4 - Regularization

Harvard University
Summer 2018
Instructors: Pavlos Protopapas, Kevin Rader

INSTRUCTIONS

- To submit your assignment follow the instructions given in canvas.
- Restart the kernel and run the whole notebook again before you submit.
- If you submit individually and you have worked with someone, please include the name of your [one] partner below.

Names of people you have worked with goes here:

In [281]:
```
from IPython.core.display import HTML
def css_styling(): styles = open("cs109.css", "r").read(); return HTML(styles)
css_styling()
```

Out[281]:

import these libraries

In [282]:
```
import warnings
warnings.filterwarnings('ignore')
import numpy as np
import pandas as pd
import matplotlib
import matplotlib.pyplot as plt
from sklearn.metrics import r2_score
from sklearn.preprocessing import PolynomialFeatures
from sklearn.linear_model import Ridge
from sklearn.linear_model import Lasso
from sklearn.linear_model import RidgeCV
from sklearn.linear_model import LassoCV
from sklearn.linear_model import LinearRegression
from sklearn.preprocessing import StandardScaler

from sklearn.model_selection import cross_val_score
from sklearn.model_selection import LeaveOneOut
from sklearn.model_selection import KFold

#import statsmodels.api as sm

from pandas.core import datetools
%matplotlib inline
```

Continuing Bike Sharing Usage Data

In this homework, we will focus on regularization and cross validation. We will continue to build regression models for the Capital Bikeshare program in Washington D.C. See homework 3 for more information about the Capital Bikeshare data that we'll be using extensively.

Data Preparation

Question 1

In HW3 Questions 1-3, you preprocessed the data in preparation for your regression analysis. We ask you to repeat those steps (particularly those in Question 3) so that we can compare the analysis models in this HW with those you developed in HW3. In this HW we'll be using models from sklearn exclusively (as opposed to statsmodels)

1.1 [From HW3] Read `data/BSS_train.csv` and `data/BSS_test.csv` into dataframes `BSS_train` and `BSS_test`, respectively. Remove the `dteday` column from both train and test dataset. We do not need it, and its format cannot be used for analysis. Also remove the `casual` and `registered` columns for both training and test datasets as they make `count` trivial.

1.2 Since we'll be exploring Regularization and Polynomial Features, it will make sense to standardize our data. Standardize the numerical features. Store the dataframes for the processed training and test predictors into the variables `X_train` and `X_test`. Store the appropriately shaped numpy arrays for the corresponding train and test `count` columns into `y_train` and `y_test`.

1.3 Use the `LinearRegression` library from `sklearn` to fit a multiple linear regression model to the training set data in `X_train`. Store the fitted model in the variable `BikeOLSModel`.

1.4 What are the training and test set R^2 scores? Store the training and test R^2 scores of the `BikeOLSModel` in a dictionary `BikeOLS_r2scores` using the string 'training' and 'test' as keys.

1.5 We're going to use bootstrapped confidence intervals (use 500 bootstrap iterations) to determine which of the estimated coefficients for the `BikeOLSModel` are statistically significant at a significance level of 5% . We'll do so by creating 3 different functions:

1. `make_bootstrap_sample(dataset_X, dataset_y)` returns a bootstrap sample of `dataset_X` and `dataset_y`
2. `calculate_coefficients(dataset_X, dataset_y, model)` returns in the form of a dictionary regression coefficients calculated by your model on `dataset_X` and `dataset_y`. The keys for regression coefficients dictionary should be the names of the features. The values should be the coefficient values of that feature calculated on your model. An example would be {'hum': 12.3, 'windspeed': -1.2, 'Sunday': 0.6 ... }
3. `get_significant_predictors(regression_coefficients, significance_level)` takes as input a list of regression coefficient dictionaries (each one the output of `calculate_coefficients` and returns a python list of the feature names of the significant predictors e.g. ['Monday', 'hum', 'holiday', ...]

In the above functions `dataset_X` should always be a pandas dataframe with your features, `dataset_y` a numpy column vector with the values of the response variable and collectively they form the dataset upon which the operations take place. `model` is the `sklearn` regression model that will be used to generate the regression coefficients. `regression_coefficients` is a list of dictionaries of numpy arrays with each numpy array containing the regression coefficients (not including the intercept) calculated from one bootstrap sample. `significance_level` represents the significance level as a floating point number. So a 5% significance level should be represented as 0.05.

Store the feature names as a list of strings in the variable `BikeOLS_significant_bootstrap` and print them for your answer.

Answers

1.1 Read `data/BSS_train.csv` and `data/BSS_test.csv` into Pandas DataFrames

In [283]:
```
# Read in data
BSS_train = pd.read_csv("data/BSS_train.csv")
BSS_test = pd.read_csv("data/BSS_test.csv")

# Drop trivial columns from the dataframes
BSS_train = BSS_train.drop(["dteday", "casual", "registered", "Unnamed: 0"], a
xis = 1)
BSS_test = BSS_test.drop(["dteday", "casual", "registered", "Unnamed: 0"], axi
s = 1)

# Print the head to check
BSS_train.head()
```

Out[283]:

	hour	holiday	year	workingday	temp	atemp	hum	windspeed	counts	spring	...	De
0	0	0	0	0	0.24	0.2879	0.81	0.0	16	0	...	0
1	1	0	0	0	0.22	0.2727	0.80	0.0	40	0	...	0
2	2	0	0	0	0.22	0.2727	0.80	0.0	32	0	...	0
3	3	0	0	0	0.24	0.2879	0.75	0.0	13	0	...	0
4	4	0	0	0	0.24	0.2879	0.75	0.0	1	0	...	0

5 rows × 32 columns

1.2 Standardizing our data

In [284]:
```
# Declare StandardScaler() object
scaler = StandardScaler()

# Initialize copy of the train and test data, store in X_train and X_test
X_train = BSS_train.copy().drop(["counts"], axis = 1)
X_test = BSS_test.copy().drop(["counts"], axis = 1)


# Iterate through each numerical feature in X_train and X_test
for col in BSS_train[["temp", "atemp", "hum", "windspeed"]].columns:

    # Standardize the features
    X_train[col] = scaler.fit_transform(X_train[[col]])
    X_test[col] = scaler.fit_transform(X_test[[col]])

# Store a copy of the response variables in y_train and y_test
y_train = BSS_train.copy()["counts"]
y_test = BSS_test.copy()["counts"]

# Check the feature variables
X_test.head()
```

Out[284]:

	hour	holiday	year	workingday	temp	atemp	hum	windspeed	spring	s
0	6	0	0	0	-1.448124	-1.192685	0.937441	-1.539692	0	0
1	9	0	0	0	-0.927233	-0.750546	0.729191	-1.539692	0	0
2	20	0	0	0	-0.510521	-0.397067	1.301879	0.511999	0	0
3	10	0	0	0	-0.718877	-0.750546	0.989503	0.271004	0	0
4	12	0	0	0	-0.718877	-0.839207	0.208565	0.874300	0	0

5 rows × 31 columns

1.3 Use the `LinearRegression` library from `sklearn` to fit a multiple linear regression.

In [285]:
```
# Declare LinearRegression object
regression = LinearRegression()

# Fit a multiple linear regression with the training data
BikeOLSModel = regression.fit(X_train, y_train)
```

1.4 What are the training and test set R^2 scores? Store the R^2 scores of the `BikeOLSModel` on the training and test sets in a dictionary `BikeOLS_r2scores`.

In [286]:
```
# Store the predicted values for the training and test data
yhat_train = regression.predict(X_train)
yhat_test = regression.predict(X_test)

# Calculate the R^2 scores for the training and test data
r2_train = r2_score(y_train, yhat_train)
r2_test = r2_score(y_test, yhat_test)

# Store the R^2 scores in a dicionary
BikeOLS_r2scores = {'train' : r2_train, 'test': r2_test }

# Print
BikeOLS_r2scores
```

Out[286]:
```
{'train': 0.4065387827969087, 'test': 0.40540416900870035}
```

1.5 We're going to use bootstrapped confidence intervals to determine which of the estimated coefficients ...

In [287]:
```
# dataset_x should be a pandas dataframe

## accepts dataset inputs as numpy arrays
def make_bootstrap_sample(dataset_X, dataset_y, size = None):
    """Returns a bootstrap sample of the given data and size

    Args:
        dataset_X: the Pandas dataframe of feature variables and values
        dataset_y: the column vector of response values
        size: the size of the Bootstrap sample with None as the default

    Returns:
        Returns a tuple containing a dataframe containing the Bootstrap X-valu
es and a list of Bootstrap y-values
    """

    # by default return a bootstrap sample of the same size as the original da
taset
    if not size: size = len(dataset_X)

    # if the X and y datasets aren't the same size, raise an exception
    if len(dataset_X) != len(dataset_y):
        raise Exception("Data size must match between dataset_X and dataset_y"
)

    # Store an ndarray of random indices
    inds_to_sample = np.random.choice(dataset_X.shape[0], size, replace = True
)

    # Store the observations located at these indices in the Dataframe as the
 bootstrap sample data
    bootstrap_dataset_X = dataset_X.iloc[inds_to_sample]
    bootstrap_dataset_y = dataset_y[inds_to_sample]

    # return as a tuple the bootstrap samples of dataset_X as a pandas datafra
me
    # and the bootstrap samples of dataset y as a numpy column vector
    return (bootstrap_dataset_X, bootstrap_dataset_y)

def calculate_coefficients(dataset_X, dataset_y, model):
    """Returns the Beta-coefficients of the given data fitted to the given mod
el

    Args:
        dataset_X: the Pandas dataframe of feature variables and values
        dataset_y: the column vector of response values
        model: the model object to be fitted with the given data

    Returns:
        coefficients_dictionary: a dictionary containing the feature names as
 keys and the
        coefficients as values."""

    # Fit the data to the given model parameter
    model.fit(dataset_X, dataset_y)
```

```
    # Initalize dictionary of coefficents
    coefficients_dictionary = {}

    # Iterate through dataset coefficients and store them in the dictionary
    for index, val in enumerate(model.coef_):
        coefficients_dictionary[dataset_X.columns[index]] = val

    # return coefficients  in the variable  coefficients_dictionary as a dicti
onary
    # with the key being the name of the feature as a string
    # the value being the value of the coefficients
    # do not return the intercept as part of this
    return coefficients_dictionary

def get_significant_predictors(regression_coefficients, significance_level):
    """Returns the significant predictors from a model

    Args:
        regression_coefficients: a list of dictionaries containing the regress
ion coefficients
        of models made from different samples
        significance_level: the alpha-level used to determine feature signific
ance

    Returns:
        significant_coefficients: a list of the significant feature names
        """

    # initialize a DataFrame to contain a list of the column names
    coefficient_values = pd.DataFrame(index=list(regression_coefficients[0].ke
ys()))

    # Store all of the values from each bootstrap sample in a column
    for row, cur_sample in enumerate(regression_coefficients):
        coefficient_values[row] = cur_sample.values()

    # Transpose the DataFrame so that each row now represents one Bootstrap sa
mple
    coefficient_values = coefficient_values.T

    # Initialize significant_coefficients to an empty list
    significant_coefficients = []

    # Store the upper and lower percentile bounds
    lower_bound = significance_level/2 * 100
    upper_bound = 100 - lower_bound

    # Iterate through all of coefficient_values and determine which coefficien
ts are significant
    for index_column, col in enumerate(coefficient_values):
        if not(((np.percentile(coefficient_values[col], lower_bound ) < 0)) &
((np.percentile(coefficient_values[col], upper_bound) > 0))):
            significant_coefficients.append(coefficient_values.columns[index_c
olumn])

    # return significant_coefficients
    return significant_coefficients
```

```
# Initialize the list of coefficients
boot_coefs = [np.nan]*500

# Create 500 Bootstrap samples
for i in range(500):
    boot_x, boot_y = make_bootstrap_sample(X_train, y_train)
    boot_coefs[i] = calculate_coefficients(boot_x, boot_y, LinearRegression())

# Store the significant predictors
BikeOLS_significant_bootstrap = get_significant_predictors(boot_coefs, 0.05)

# Print the significant predictors
print("\033[1mSignificant Predictors:\033[0m\n{0}".format(BikeOLS_significant_
bootstrap))
```

Significant Predictors:

```
['hour', 'holiday', 'year', 'workingday', 'temp', 'hum', 'windspeed', 'sprin
g', 'summer', 'fall', 'Mar', 'Apr', 'May', 'Jun', 'Jul', 'Aug', 'Oct', 'Nov',
 'Dec', 'Sat', 'Cloudy', 'Snow', 'Storm']
```

Penalization Methods

In HW 3 Question 5 we explored using subset selection to find a significant subset of features. We then fit a regression model just on that subset of features instead of on the full dataset (including all features). As an alternative to selecting a subset of predictors and fitting a regression model on the subset, one can fit a linear regression model on all predictors, but shrink or regularize the coefficient estimates to make sure that the model does not "overfit" the training set.

Question 2

We're going to use Ridge and Lasso regression regularization techniques to fit linear models to the training set. We'll use cross-validation and shrinkage parameters λ from the set $\{.001, .005, 1, 5, 10, 50, 100, 500, 1000\}$ to pick the best model for each regularization technique.

2.1 Use 5-fold cross-validation to pick the best shrinkage parameter from the set $\{.001, .005, 1, 5, 10, 50, 100, 500, 1000\}$for your Ridge Regression model on the training data. Fit a Ridge Regression model on the training set with the selected shrinkage parameter and store your fitted model in the variable `BikeRRModel`. Store the selected shrinkage parameter in the variable `BikeRR_shrinkage_parameter`.

2.2 Use 5-fold cross-validation to pick the best shrinkage parameter from the set $\{.001, .005, 1, 5, 10, 50, 100, 500, 1000\}$for your Lasso Regression model on the training data. Fit a Lasso Regression model on the training set with the selected shrinkage parameter and store your fitted model in the variable `BikeLRModel`. Store the selected shrinkage parameter in the variable `BikeLR_shrinkage_parameter`.

2.3 Create three dictionaries `BikeOLSparams`, `BikeLRparams`, and `BikeRRparams`. Store in each the corresponding regression coefficients for each of the regression models indexed by the string feature name.

2.4 For the Lasso and Ridge Regression models list the features that are assigned a coefficient value close to 0 (i.e. the absolute value of the coefficient is less than 0.1). How closely do they match the redundant predictors found (if any) in HW 3, Question 5?

2.5 To get a visual sense of how the features different regression models (Multiple Linear Regression, Ridge Regression, Lasso Regression) estimate coefficients, order the features by magnitude of the estimated coefficients in the Multiple Linear Regression Model (no shrinkage). Plot a bar graph of the magnitude (absolute value) of the estimated coefficients from Multiple Linear Regression in order from greatest to least. Using a different color (and alpha values) overlay bar graphs of the magnitude of the estimated coefficients (in the same order as the Multiple Linear Regression coefficients) from Ridge and Lasso Regression.

2.6 Let's examine a pair of features we believe to be related. Is there a difference in the way Ridge and Lasso regression assign coefficients to the predictors `temp` and `atemp`? If so, explain the reason for the difference.

2.7 Discuss the Results:

1. How do the estimated coefficients compare to or differ from the coefficients estimated by a plain linear regression (without shrinkage penalty) in Question 1?
2. Is there a difference between coefficients estimated by the two shrinkage methods? If so, give an explantion for the difference.
3. Is the significance related to the shrinkage in some way?

Hint: You may use `sklearn`'s `RidgeCV` and `LassoCV` classes to implement Ridge and Lasso regression. These classes automatically perform cross-validation to tune the parameter λ from a given range of values.

Answers

In [288]:
```
lambdas = [.001, .005, 1, 5, 10, 50, 100, 500, 1000]
```

2.1 Use 5-fold cross-validation to pick the best shrinkage parameter from the set $\{.001, .005, 1, 5, 10, 50, 100, 500, 1000\}$ for your Ridge Regression model.

In [289]:
```
# Initialize a 5-fold KFOLD object
kfold5 = KFold(5, shuffle=True)

# Initialize a Ridge cross-validation object
ridgeCV_object = RidgeCV(alphas=(lambdas), cv=kfold5)

# Fit the training data with the CV object
BikeRRModel = ridgeCV_object.fit(X_train, y_train)

# Store the optimum shrinkage parameter according to the algorithm
BikeRR_shrinkage_parameter = BikeRRModel.alpha_
```

2.2 Use 5-fold cross-validation to pick the best shrinkage parameter from the set $\{.001, .005, 1, 5, 10, 50, 100, 500, 1000\}$ for your Lasso Regression model.

In [290]:
```
# Initialize a Lasso cross-validation object
lassoCV_object = LassoCV(alphas= lambdas, cv=kfold5)

# Fit the training data with the CV object
BikeLRModel = lassoCV_object.fit(X_train, y_train)

# Store the optimum shrinkage parameter according to the algorithm
BikeLR_shrinkage_parameter = BikeLRModel.alpha_
```

2.3 Create three dictionaries `BikeOLSparams`, `BikeLRparams`, and `BikeRRparams`. Store in each the corresponding regression coefficients.

In [291]:
```
# Create the three dictionaries using dict() and zip() functions
BikeLRparams = dict(zip(X_train.columns.values, BikeLRModel.coef_))
BikeRRparams = dict(zip(X_train.columns.values, BikeRRModel.coef_))
BikeOLSparams = dict(zip(X_train.columns.values, BikeOLSModel.coef_))
```

2.4 For the Lasso and Ridge Regression models list the features that are assigned a coefficient value close to 0 ...

In [292]:
```
# Use list comprehenion to create a list of features with coefficient values c
lose to 0
lr_small = [k for k,v in BikeLRparams.items() if abs(v) < 0.01]
rr_small = [k for k,v in BikeRRparams.items() if abs(v) < 0.01]

# Print Results
print("\033[1mFeatures with Coefficient Absolute Value < 0.01:\033[0m\nLasso:
{0}\nRidge: {1}".format(lr_small, rr_small))
```

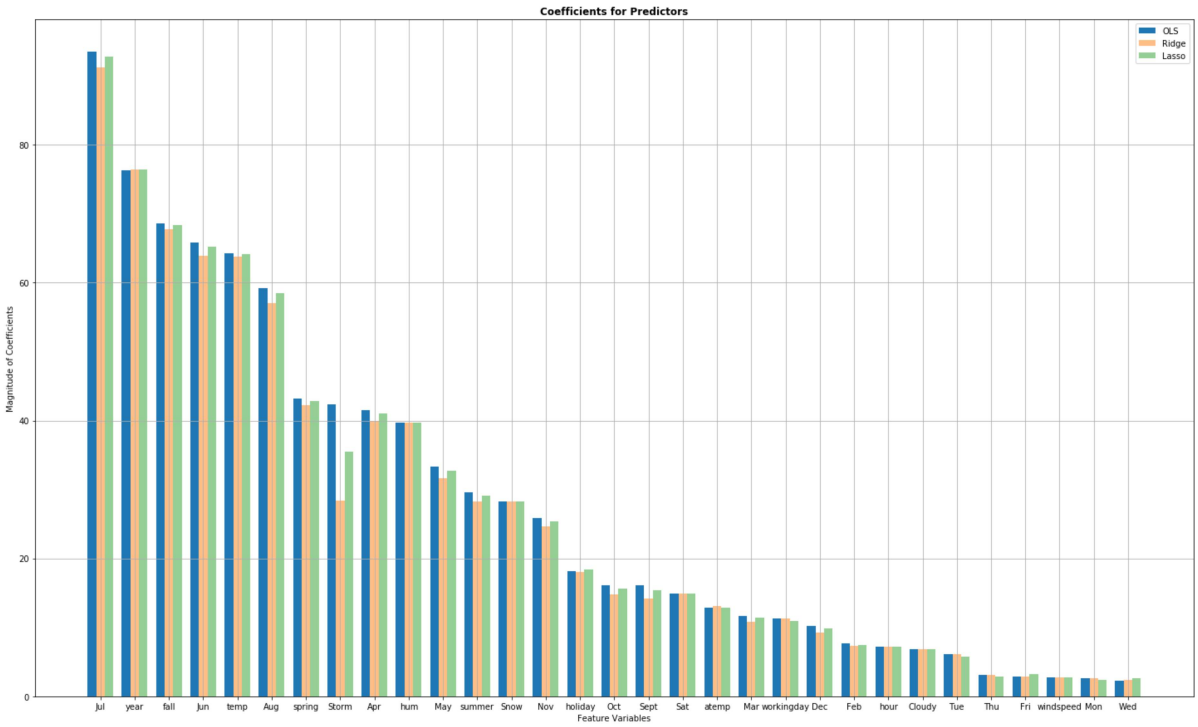
```
Features with Coefficient Absolute Value < 0.01:
Lasso: []
Ridge: []
```

How closely do they match the redundant predictors found (if any) in HW 3, Question 5?

There were no features found with coefficient values close to 0 and therefore the list has no match-up with the redundant features from HW 3 Question 5

2.5 To get a visual sense of how the features different regression models (Multiple Linear Regression, Ridge Regression, Lasso Regression) estimate coefficients, order the features by magnitude of the estimated coefficients in the Multiple Linear Regression Model (no shrinkage).

In [293]:
```
# Initialize the figure
fig, ax = plt.subplots(1,1, figsize=(25,15))

# Sort the feature variables and coefficients in BikeOLSParams by magnitude
BikeOLS_sort = sorted(BikeOLSparams.items(), key=lambda dict_key: abs(dict_key
[1]))

# Put them in order from greatest magnitude to least
BikeOLS_sort.reverse()

# Store the number of feature variables
num_features = len(BikeOLS_sort)

# Initialize empty lists to hold Ridge Regression and Lasso Regression coeffic
ients
rr_values = [np.nan] * num_features
lr_values = [np.nan] * num_features

# Store the Ridge and Lasso coefficients in the lists
for index, (a, b) in enumerate(BikeOLS_sort):
    rr_values[index] = BikeRRparams[a]
    lr_values[index] = BikeLRparams[a]

# Store the OLS regression coefficients in a list
ols_values = list(np.array(BikeOLS_sort)[:,1])

# Store x_values and width values in the corresponding variables
x_values = np.arange(num_features)
width=1/4

# Plot OLS, Ridge, and Lasso regression data in the same bar plot, adjusting t
he width, color, alpha levels, and position
ax.bar(x_values, [abs(float(number)) for number in ols_values], width, alpha=1
, label="OLS")
ax.bar(x_values + width, [abs(number) for number in rr_values], width, alpha=
0.5, label = "Ridge")
ax.bar(x_values + width * 2, [abs(number) for number in lr_values], -width, al
pha=0.5, label = "Lasso")

# Add labels, grid, and legend
ax.set_title("Coefficients for Predictors", fontweight="bold")
ax.set_xticklabels([a for (a, b) in BikeOLS_sort])
ax.set_xticks(x_values + .25)
ax.set_xlabel("Feature Variables")
ax.set_ylabel("Magnitude of Coefficients")
ax.grid(True)
ax.legend();
```



2.6 Let's examine a pair of features we believe to be related. Is there a difference in the way Ridge and Lasso regression assign coefficients ...v

---

The coefficient for temp is much larger than the one for atemp for both Ridge an Lasso regression models. However, the ratio temp : atemp is slightly larger for the Lasso coefficients than for the Ridge coefficients.

2.7.1 How do the estimated coefficients compare to or differ from ...

The Lasso and OLS coefficients align much closer than OLS and Ridge coefficients. The following variables have the most visible discrepancies: Storm, October, Summer, Sept, May, Mar, Dec, August, June, and July. The storm coefficient has the most variation between the models (the Ridge model differs drastically while the OLS and Lasso also differ significantly). The values for the OLS coefficients are generally larger or equal in magnitude to the other two models, with the exceptions of where ridge is larger: atemp. For holiday and Friday, Lasso is larger, and for Wednesday and year both the Lasso and Ridge coefficents are larger than the OLS model.

2.7.2 Is there a difference between coefficients estimated by the two shrinkage methods ...

Yes, there is a difference. Coefficients estimated by the Lasso Regression model are generally higher in magnitude than the coefficients estimated by the Ridge and OLS regression models.

2.7.3 Is the significance related to the shrinkage in some way?

In general, Lasso has a higher estimated coefficient magnitude than Ridge since the Lasso model sets the irrelevant coefficients to zero thereby increasing the emphasis on the coefficients that actually have an impact on the model.

Question 3: Polynomial Features, Interaction Terms, and Cross Validation

We would like to fit a model to include all main effects and polynomial terms for numerical predictors up to the 4^{th} order. More precisely use the following terms:

- predictors in `X_train` and `X_test`
- X_j^1, X_j^2, X_j^3, and X_j^4 for each numerical predictor X_j

3.1 Create an expanded training set including all the desired terms mentioned above. Store that training set (as a pandas dataframe) in the variable `X_train_poly`. Create the corresponding test set and store it as a pandas dataframe in `X_test_poly`.

3.2 Discuss the following:

1. What are the dimensions of this 'design matrix' of all the predictor variables in 3.1?
2. What issues may we run into attempting to fit a regression model using all of these predictors?

3.3 Let's try fitting a regression model on all the predictors anyway. Use the `LinearRegression` library from `sklearn` to fit a multiple linear regression model to the training set data in `X_train_poly`. Store the fitted model in the variable `BikeOLSPolyModel`.

3.4 Discuss the following:

1. What are the training and test R^2 scores?
2. How does the model performance compare with the OLS model on the original set of features in Question 1?

3.5 The training set R^2 score we generated for our model with polynomial and interaction terms doesn't have any error bars. Let's use cross-validation to generate sample sets of R^2 for our model. Use 5-fold cross-validation to generate R^2 scores for the multiple linear regression model with polynomial terms. What are the mean and standard deviation of the R^2 scores for your model.

3.6 Visualize the R^2 scores generated from the 5-fold cross validation as a box and whisker plot.

3.7 We've used cross-validation to generate error bars around our R^2 scores, but another use of cross-validation is as a way of model selection. Let's construct the following model alternatives:

1. Multiple linear regression model generated based upon the feature set in Question 1 (let's call these the base features.
2. base features plus polynomial features to order 2
3. base features plus polynomial features to order 4

Use 5-fold cross validation on the training set to select the best model. Make sure to evaluate all the models as much as possible on the same folds. For each model generate a mean and standard deviation for the R^2 score.

3.8 Visualize the R^2 scores generated for each model from 5-fold cross validation in box and whiskers plots. Do the box and whisker plots influence your view of which model was best?

3.9 Evaluate each of the model alternatives on the test set. How do the results compare with the results from cross-validation?

Answers

3.1 Create an expanded training set including all the desired terms mentioned above. Store that training set (as a numpy array) in the variable `X_train_poly`....

In [294]:
```
# Create dataframe of higher order polynomials on a feature column up to order
 k
def get_polynomial_features(df, feature_column, degree):
    """Returns higher order features of selected feature columns

    Args:
        df: the data to be split and used
        feature_column: the column name to return higher order features for
        degree: the highest degree for the feature

    Returns:
        Returns a DataFrame of the higher order features for the feature varia
ble
    """

    # Initialize a PolynomialFeature model with the given degree
    poly_model = PolynomialFeatures(degree, include_bias=False)

    # Store the feature data in a DataFrame
    feature_data = df[feature_column]

    # transform to get all the polynomial features of this column
    higher_order_features = poly_model.fit_transform(feature_data.values.resha
pe(-1,1))

    # Extract feature names for the higher order features
    feature_names = poly_model.get_feature_names([feature_column])

    # Return the new feature data as a dataframe
    return pd.DataFrame(higher_order_features[:,1:], columns = feature_names[1
:])

# define continuous features
continuous_columns = ['temp', 'atemp', 'hum', 'windspeed']

# generate dfs with the higher order features
quartic_train =[get_polynomial_features(X_train, feature, 4) for feature in co
ntinuous_columns]
quartic_test = [get_polynomial_features(X_test, feature, 4) for feature in con
tinuous_columns]

# append the higher order features to the other feature data
quartic_train = pd.concat(quartic_train, axis=1)
quartic_test = pd.concat(quartic_test, axis=1)

# Stor the list of column names
quartic_columns = quartic_train.columns

# scale higher order polynomial features
scaler = StandardScaler().fit(quartic_train)

# Standardize the features
quartic_train[quartic_columns] = scaler.transform(quartic_train)
quartic_test[quartic_columns] = scaler.transform(quartic_test)

# concatenate to get the full data frames
```

```
X_train_poly = pd.concat([X_train, quartic_train], axis=1)
X_test_poly = pd.concat([X_test, quartic_test], axis=1);
```

In [295]:
```
# Print the feature data shape
X_train_poly.shape
```

Out[295]: (13903, 43)

3.2.1 What are the dimensions of this 'design matrix'...**

There are 13903 rows corresponding to 13903 observations in the training set. There are 43 columns in the design matrix corresponding to the 31 original feature variables as well as the quadratic, cubic, and quartic features for the four continuous feature variables.

3.2.2 What issues may we run into attempting to fit a regression model using all of these predictors? ...**

With a high number of feature variables and feature variables of 4 different degrees for numerous variables, overfitting is a potential issue. With feature variables that also seem to be related (i.e. month and season, atemp and temp, workingday and days of the week), collinearity seems likely to be a potential issue with this model.

3.3 Let's try fitting a regression model on all the predictors anyway. Use the `LinearRegression` library from `sklearn` to fit a multiple linear regression model ....

In [296]:
```
# store the fitted model
BikeOLSPolyModel = LinearRegression().fit(X_train_poly, y_train)
```

3.4.1 What are the training and test R^2 scores?

In [297]:
```
# Store the training and test R^2 scores
train_r2_d4 = r2_score(y_train, BikeOLSPolyModel.predict(X_train_poly))
test_r2_d4 = r2_score(y_test, BikeOLSPolyModel.predict(X_test_poly))

# Print the results
print("\033[1mR^2 Scores:\033[0m\nTraining: {0}\nTest: {1}".format(train_r2_d4
, test_r2_d4))
```

```
R^2 Scores:
Training: 0.42230805166587104
Test: 0.4194232238324874
```

3.4.2 How does the model performance compare with the OLS model on the original set of features in Question 1?

The R^2 values for the polynomial model are over 2 % better (0.01) than the R^2 values for the OLS model in Question 1. Based on the test R^2 values, the polynomial model performs better. However, it is uncertain whether adding all 12 feature columns is beneficial, and a stepwise selection could be helpful with optimizing the model features.

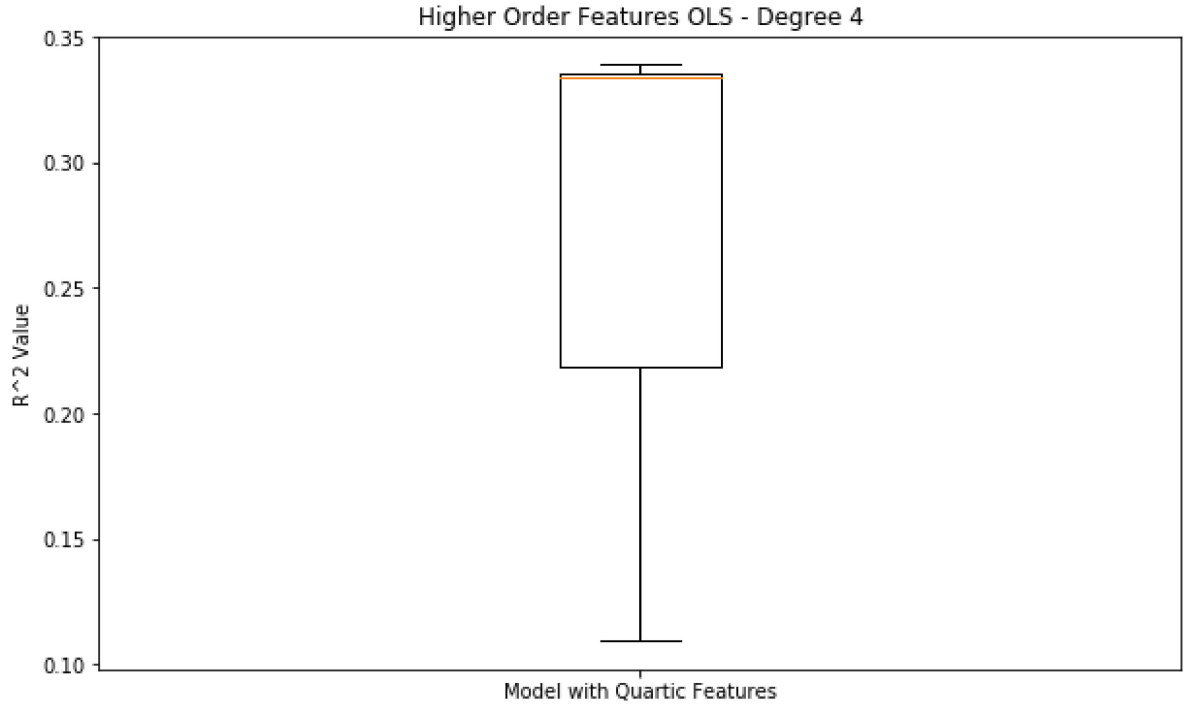
3.5 The training set R^2 score we generated for our model with polynomial and interaction terms doesn't have any error bars. Let's use cross-validation to generate sample...

In [298]:
```
# Store the cv scores for the OLS model with higher order features
cv_scores_quart = cross_val_score(LinearRegression(), X_train_poly, y_train, c
v=5)

# Print mean and standard deviation
print("\033[1mCV Scores (Quartic):\033[0m\nMean: {0}\nStandard Deviation: {1}"
.format(cv_scores_quart.mean(), cv_scores_quart.std()))
```

```
CV Scores (Quartic):
Mean: 0.26727805322296977
Standard Deviation: 0.09109366555046658
```

3.6

In [300]:
```
# Initialize the figure
fig, ax = plt.subplots(1,1,figsize=(10,6))

# Display boxplot
ax.boxplot(cv_scores_quart)

# Add labels
ax.set_title("Higher Order Features OLS - Degree 4")
ax.set_xticklabels(["Model with Quartic Features"])
ax.set_ylabel("R^2 Value");
```



3.7 We've used cross-validation to generate error bars around our R^2 scores, but another use of cross-validation is as a way of model selection. Let's construct the following model alternatives ...

In [301]:
```
# generate dfs with the quadratic features
quadratic_train =[get_polynomial_features(X_train, feature, 2) for feature in
continuous_columns]
quadratic_test = [get_polynomial_features(X_test, feature, 2) for feature in c
ontinuous_columns]

# append the quadratic features to the other feature data
quadratic_train = pd.concat(quadratic_train, axis=1)
quadratic_test = pd.concat(quadratic_test, axis=1)

# Store the list of column names
quadratic_columns = quadratic_train.columns

# scale quadratic features
scaler = StandardScaler().fit(quadratic_train)

# Standardize the features
quadratic_train[quadratic_columns] = scaler.transform(quadratic_train)
quadratic_test[quadratic_columns] = scaler.transform(quadratic_test)

# concatenate to get the full data frames
X_train_quad = pd.concat([X_train, quadratic_train], axis=1)
X_test_quad = pd.concat([X_test, quadratic_test], axis=1)

# Store cv_scores for the data from Question 1
cv_scores_quad = cross_val_score(LinearRegression(), X_train_quad, y_train, cv
=5)

# Print mean and standard deviation
print("\033[1mCV Scores (Quadratic):\033[0m\nMean: {0}\nStandard Deviation:
{1}".format(cv_scores_quad.mean(), cv_scores_quad.std()))
```

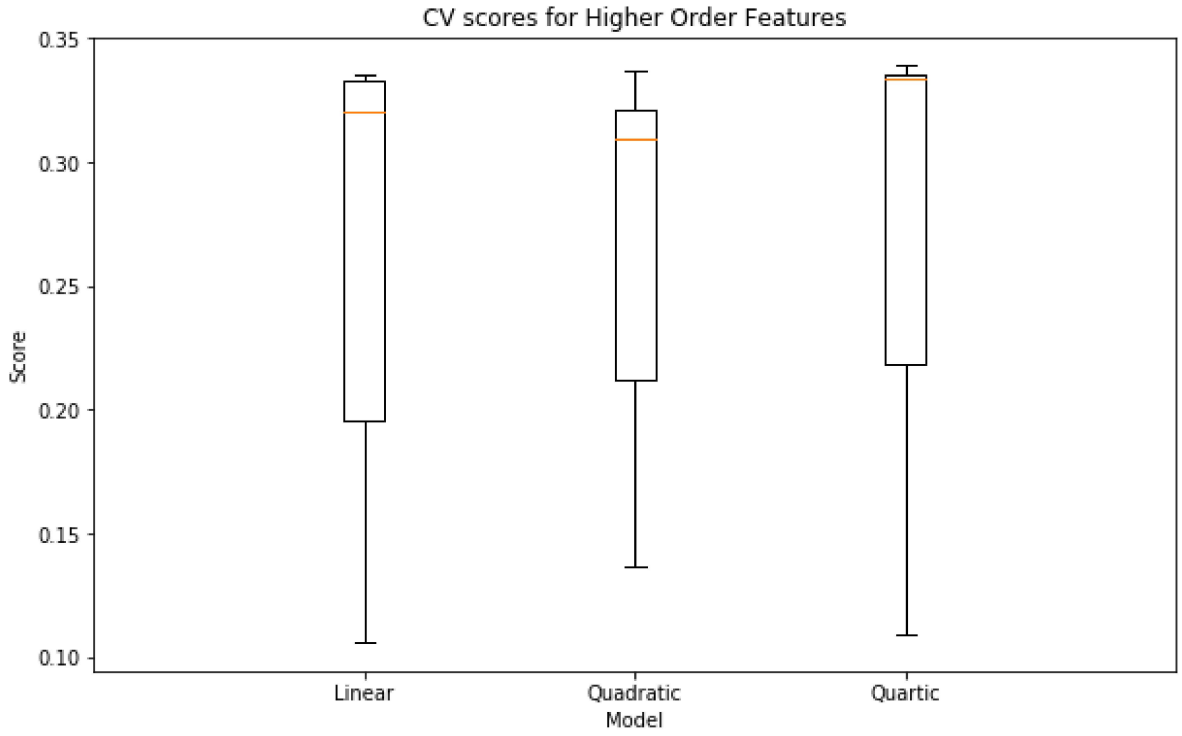
```
CV Scores (Quadratic):
Mean: 0.263258751169427
Standard Deviation: 0.07670352483743688
```

In [302]:
```
# Store cv_scores for the data from Question 1
cv_scores = cross_val_score(LinearRegression(), X_train, y_train, cv=5)

# Print mean and standard deviation
print("\033[1mCV Scores (Linear):\033[0m\nMean: {0}\nStandard Deviation: {1}".
format(cv_scores.mean(), cv_scores.std()))
```

```
CV Scores (Linear):
Mean: 0.2579944364362185
Standard Deviation: 0.09220055096227689
```

3.8 Visualize the R^2 scores generated for each model from 5-fold cross validation in box and whiskers plots. Do the box and whisker plots influence your view of which model was best? ...

In [303]:
```
# Initialize the figure
fig, ax = plt.subplots(1,1, figsize=(10,6))

# Plot the Boxplots of CV scores
cv_scores_list = [cv_scores, cv_scores_quad, cv_scores_quart]
for i, score in enumerate(cv_scores_list):
    ax.boxplot(score, positions=[i])

# Add Labels
ax.set_title("CV scores for Higher Order Features")
ax.set_xlabel("Model")
ax.set_ylabel("Score")
ax.set_xlim(-1,3)
ax.set_xticks(range(3))
ax.set_xticklabels(["Linear","Quadratic","Quartic"]);
```



Looking at the box and whisker plots, the quadratic model seems to be the best one as the variance in score appears to be the smallest and most normally distributed of the three models. However, all three models appear to be highly skewed to the left, with the quartic being the most extreme of the three.

3.9 Evaluate each of the model alternatives on the test set. How do the results compare with the results from cross-validation?

In [304]:
```
# Fit the quadratic features to a model and obtain the test R^2 score
quadratic_model = LinearRegression().fit(X_train_quad, y_train)
test_r2_d2 = r2_score(y_test, quadratic_model.predict(X_test_quad))

# Call R^2 test variable for model from Q1
test_r2_d1 = r2_test

# Print the Test R^2 Scores
print("\033[1mTest R^2 scores:\033[0m\nQuartic Features: {0}\nQuadratic Featur
es: {1}\nLinear Features: {2}".format(test_r2_d4, test_r2_d2, test_r2_d1))
```

```
Test R^2 scores:
Quartic Features: 0.4194232238324874
Quadratic Features: 0.4098337591596225
Linear Features: 0.40540416900870035
```

The results here would potentially lead to a different conclusion than the one made with Cross Validation, as the Quartic Features seems to fit the test data decently better than the other two models based on test R^2 values alone.
