cs109a_hw4_109

October 21, 2018

1 CS109A Introduction to Data Science:

1.1 Homework 4 - Regularization

Harvard University Fall 2018 Instructors: Pavlos Protopapas, Kevin Rader

1.1.1 INSTRUCTIONS

- **This homework must be completed individually.**
- To submit your assignment follow the instructions given in Canvas.
- Restart the kernel and run the whole notebook again before you submit.
- As much as possible, try and stick to the hints and functions we import at the top of the homework, as those are the ideas and tools the class supports and is aiming to teach. And if a problem specifies a particular library you're required to use that library, and possibly others from the import list.

Names of people you have worked with goes here:

```
In [35]: #RUN THIS CELL
         import requests
         from IPython.core.display import HTML
         styles = requests.get("https://raw.githubusercontent.com/Harvard-IACS/2018-CS109A/mas
         HTML(styles)
```

```
Out[35]: <IPython.core.display.HTML object>
```

import these libraries

```
In [36]: 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 train_test_split

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 statsmodels.regression.linear_model import OLS

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

2 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.

Question 1 [20pts] Data pre-processing

1.1 Read in the provided `bikes_student.csv` to a data frame named `bikes_main`. Split it into a training set `bikes_train` and a validation set `bikes_val`. Use `random_state=90`, a test set size of .2, and stratify on month. Remember to specify the data's index column as you read it in.

1.2 As with last homework, the response will be the `counts` column and we'll drop `counts`, `registered` and `casual` for being trivial predictors, drop `workingday` and `month` for being multicollinear with other columns, and `dteday` for being inappropriate for regression. Write code to do this.

Encapsulate this process as a function with appropriate inputs and outputs, and **test** your code by producing `practice_y_train` and `practice_X_train`.

1.3 Write a function to standardize a provided subset of columns in your training/validation/test sets. Remember that while you will be scaling all of your data, you must learn the scaling parameters (mean and SD) from only the training set.

Test your code by building a list of all non-binary columns in your `practice_X_train` and scaling only those columns. Call the result `practice_X_train_scaled`. Display the `.describe()` and verify that you have correctly scaled all columns, including the polynomial columns.

Hint: employ the provided list of binary columns and use `pd.columns.difference()`

```
binary_columns = [ 'holiday', 'workingday','Feb', 'Mar', 'Apr',        'May',
'Jun', 'Jul', 'Aug', 'Sept', 'Oct', 'Nov', 'Dec', 'spring',        'summer',
'fall', 'Mon', 'Tue', 'Wed', 'Thu', 'Fri', 'Sat',        'Cloudy', 'Snow',
'Storm']
```

1.4 Write a code to augment your a dataset with higher-order features for `temp`, `atemp`, `hum`,`windspeed`, and `hour`. You should include ONLY the pure powers of these columns. So with degree=2 you should produce `atemp^2` and `hum^2` but not `atemp*hum` or any other two-feature interactions.

Encapsulate this process as a function with appropriate inputs and outputs, and test your code by producing `practice_X_train_poly`, a training dataset with quadratic and cubic features built from `practice_X_train_scaled`, and printing `practice_X_train_poly`'s column names and `.head()`.

1.5 Write code to add interaction terms to the model. Specifically, we want interactions between the continuous predictors (`temp`,`atemp`, `hum`,`windspeed`) and the month and weekday dummies (`Feb`, `Mar`...`Dec`, `Mon`, `Tue`, ... `Sat`). That means you SHOULD build `atemp*Feb` and `hum*Mon` and so on, but NOT `Feb*Mar` and NOT `Feb*Tue`. The interaction terms should always be a continuous feature times a month dummy or a continuous feature times a weekday dummy.

Encapsulate this process as a function with appropriate inputs and outputs, and test your code by adding interaction terms to `practice_X_train_poly` and show its column names and `.head()`**

1.6 Combine all your code so far into a function that takes in `bikes_train`, `bikes_val`, the names of columns for polynomial, the target column, the columns to be dropped and produces computation-ready design matrices `X_train` and `X_val` and responses `y_train` and `y_val`. Your final function should build correct, scaled design matrices with the stated interaction terms and any polynomial degree.

2.0.1 Solutions

1.1 Read in the provided `bikes_student.csv` to a data frame named `bikes_main`. Split it into a training set `bikes_train` and a validation set `bikes_val`. Use `random_state=90`, a test set size of .2, and stratify on month. Remember to specify the data's index column as you read it in.

```
In [37]: bikes_main = pd.read_csv('data/bikes_student.csv', sep=",", index_col = 'Unnamed: 0')
         bikes_main.head()
         bikes_main.shape

Out[37]: (1250, 36)

In [38]: bikes_train, bikes_val = train_test_split(bikes_main, test_size = 0.2, random_state =
                                                   stratify=bikes_main['month'])
         bikes_train.shape
         bikes_val.shape

Out[38]: (250, 36)
```

1.2 As with last homework, the response will be the `counts` column and we'll drop `counts`, `registered` and `casual` for being trivial predictors, drop `workingday` and `month` for being multicolinear with other columns, and `dteday` for being inappropriate for regression. Write code to do this.

Encapsulate this process as a function with appropriate inputs and outputs, and test your code by producing `practice_y_train` and `practice_X_train`

```
In [39]: def drop_trivials(df, drop_columns = None):
             df = df.drop(drop_columns, axis=1)
             return df

         columns = ['counts', 'registered', 'casual', 'workingday', 'month', 'dteday']
```

```
        practice_y_train = bikes_train['counts']
        practice_X_train = drop_trivials(bikes_train, columns)

        display(practice_y_train.head(), practice_X_train.head())
```

```
15762    111
4213     170
14301     16
15900     24
14320    306
Name: counts, dtype: int64
```

	hour	year	holiday	temp	atemp	hum	windspeed	Feb	Mar	Apr	\
15762	23	1	0	0.54	0.5152	0.73	0.1045	0	0	0	
4213	11	0	0	0.76	0.6667	0.35	0.2239	0	0	0	
14301	2	1	0	0.66	0.6212	0.69	0.0000	0	0	0	
15900	5	1	0	0.30	0.3030	0.81	0.1343	0	0	0	
14320	21	1	0	0.70	0.6515	0.61	0.1642	0	0	0	

	...	fall	Mon	Tue	Wed	Thu	Fri	Sat	Cloudy	Snow	Storm
15762	...	1	0	1	0	0	0	0	0	0	0
4213	...	0	0	0	1	0	0	0	0	0	0
14301	...	0	0	0	0	0	1	0	0	0	0
15900	...	1	0	0	1	0	0	0	1	0	0
14320	...	0	0	0	0	0	1	0	1	0	0

```
[5 rows x 30 columns]
```

1.3 Write a function to standardize a provided subset of columns in your training/validation/test sets. Remember that while you will be scaling all of your data, you must learn the scaling parameters (mean and SD) from only the training set.

Test your code by building a list of all non-binary columns in your `practice_X_train` and scaling only those columns. Call the result `practice_X_train_scaled`. Display the `.describe()` and verify that you have correctly scaled all columns, including the polynomial columns.

Hint: employ the provided list of binary columns and use `pd.columns.difference()`

```
binary_columns = [ 'holiday', 'workingday','Feb', 'Mar', 'Apr',        'May',
'Jun', 'Jul', 'Aug', 'Sept', 'Oct', 'Nov', 'Dec', 'spring',        'summer',
'fall', 'Mon', 'Tue', 'Wed', 'Thu', 'Fri', 'Sat',        'Cloudy', 'Snow',
'Storm']
```

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

```
        def standardize(df, binary_columns):
            #Get the subset
            all_predictors = df.columns.difference(binary_columns)
```

```
    #learn the parameters from the train set
    scaler = StandardScaler().fit(df[all_predictors])

    #scale the practice set in place
    df[all_predictors] = scaler.transform(df[all_predictors])
    return df


binary_columns = [ 'holiday', 'workingday','Feb', 'Mar', 'Apr',
           'May', 'Jun', 'Jul', 'Aug', 'Sept', 'Oct', 'Nov', 'Dec', 'spring',
           'summer', 'fall', 'Mon', 'Tue', 'Wed', 'Thu', 'Fri', 'Sat',
           'Cloudy', 'Snow', 'Storm']
practice_X_train_scaled = standardize(practice_X_train, binary_columns)
practice_X_train_scaled.describe()
```

Out[40]:

	hour	year	holiday	temp	atemp	\
count	1.000000e+03	1.000000e+03	1000.000000	1.000000e+03	1.000000e+03	
mean	-1.994516e-16	2.686740e-17	0.027000	3.019807e-17	-1.256772e-16	
std	1.000500e+00	1.000500e+00	0.162164	1.000500e+00	1.000500e+00	
min	-1.646163e+00	-1.018165e+00	0.000000	-2.347976e+00	-2.402605e+00	
25%	-9.189949e-01	-1.018165e+00	0.000000	-7.922693e-01	-8.121270e-01	
50%	-4.639332e-02	9.821591e-01	0.000000	3.744066e-02	7.147176e-02	
75%	8.262083e-01	9.821591e-01	0.000000	8.671507e-01	8.670022e-01	
max	1.698810e+00	9.821591e-01	1.000000	2.319143e+00	2.546131e+00	

	hum	windspeed	Feb	Mar	Apr	\
count	1.000000e+03	1.000000e+03	1000.000000	1000.000000	1000.000000	
mean	5.995204e-17	1.301181e-16	0.078000	0.085000	0.082000	
std	1.000500e+00	1.000500e+00	0.268306	0.279021	0.274502	
min	-3.397602e+00	-1.554205e+00	0.000000	0.000000	0.000000	
25%	-7.421467e-01	-7.231056e-01	0.000000	0.000000	0.000000	
50%	5.448995e-02	-1.130295e-02	0.000000	0.000000	0.000000	
75%	8.511266e-01	4.634972e-01	0.000000	0.000000	0.000000	
max	1.913309e+00	5.211499e+00	1.000000	1.000000	1.000000	

	...	fall	Mon	Tue	Wed	\
count	...	1000.000000	1000.000000	1000.000000	1000.000000	
mean	...	0.248000	0.143000	0.148000	0.162000	
std	...	0.432068	0.350248	0.355278	0.368635	
min	...	0.000000	0.000000	0.000000	0.000000	
25%	...	0.000000	0.000000	0.000000	0.000000	
50%	...	0.000000	0.000000	0.000000	0.000000	
75%	...	0.000000	0.000000	0.000000	0.000000	
max	...	1.000000	1.000000	1.000000	1.000000	

	Thu	Fri	Sat	Cloudy	Snow	Storm
count	1000.000000	1000.00000	1000.00000	1000.000000	1000.000000	1000.0
mean	0.128000	0.12700	0.15000	0.280000	0.082000	0.0

```
std       0.334257     0.33314     0.35725     0.449224     0.274502     0.0
min       0.000000     0.00000     0.00000     0.000000     0.000000     0.0
25%       0.000000     0.00000     0.00000     0.000000     0.000000     0.0
50%       0.000000     0.00000     0.00000     0.000000     0.000000     0.0
75%       0.000000     0.00000     0.00000     1.000000     0.000000     0.0
max       1.000000     1.00000     1.00000     1.000000     1.000000     0.0

[8 rows x 30 columns]
```

1.4 Write a code to augment your a dataset with higher-order features for `temp`, `atemp`, `hum`,`windspeed`, and `hour`. You should include ONLY pure powers of these columns. So with degree=2 you should produce `atemp^2` and `hum^2` but not `atemp*hum` or any other two-feature interactions.

Encapsulate this process as a function with apropriate inputs and outputs, and test your code by producing `practice_X_train_poly`, a training dataset with qudratic and cubic features built from `practice_X_train_scaled`, and printing `practice_X_train_poly`'s column names and `.head()`.

```
In [41]: def df_augment(df, degree, columns):
             for c in columns:
                 for i in range(2, degree+1):
                     df[c + '_'+ str(i)] = df[c]**i
             return df

         columns = ['temp', 'atemp', 'hum','windspeed', 'hour']
         higher_order = 3
         practice_X_train_poly = df_augment(practice_X_train_scaled, higher_order, columns)
         display(practice_X_train_poly.describe())
```

```
                hour          year      holiday          temp         atemp  \
count  1.000000e+03  1.000000e+03  1000.000000  1.000000e+03  1.000000e+03
mean  -1.994516e-16  2.686740e-17     0.027000  3.019807e-17 -1.256772e-16
std    1.000500e+00  1.000500e+00     0.162164  1.000500e+00  1.000500e+00
min   -1.646163e+00 -1.018165e+00     0.000000 -2.347976e+00 -2.402605e+00
25%   -9.189949e-01 -1.018165e+00     0.000000 -7.922693e-01 -8.121270e-01
50%   -4.639332e-02  9.821591e-01     0.000000  3.744066e-02  7.147176e-02
75%    8.262083e-01  9.821591e-01     0.000000  8.671507e-01  8.670022e-01
max    1.698810e+00  9.821591e-01     1.000000  2.319143e+00  2.546131e+00

                hum     windspeed          Feb          Mar          Apr  \
count  1.000000e+03  1.000000e+03  1000.000000  1000.000000  1000.000000
mean   5.995204e-17  1.301181e-16     0.078000     0.085000     0.082000
std    1.000500e+00  1.000500e+00     0.268306     0.279021     0.274502
min   -3.397602e+00 -1.554205e+00     0.000000     0.000000     0.000000
25%   -7.421467e-01 -7.231056e-01     0.000000     0.000000     0.000000
50%    5.448995e-02 -1.130295e-02     0.000000     0.000000     0.000000
75%    8.511266e-01  4.634972e-01     0.000000     0.000000     0.000000
max    1.913309e+00  5.211499e+00     1.000000     1.000000     1.000000
```

```
          ...            temp_2       temp_3      atemp_2      atemp_3  \
count     ...       1000.000000  1000.000000  1000.000000  1000.000000
mean      ...          1.000000    -0.027791     1.000000    -0.105907
std       ...          1.010902     2.365524     1.041730     2.456716
min       ...          0.001402   -12.944365     0.000275   -13.869057
25%       ...          0.231484    -0.497300     0.136927    -0.535638
50%       ...          0.751950     0.000052     0.751693     0.000365
75%       ...          1.457149     0.652054     1.489478     0.651719
max       ...          5.512989    12.473338     6.482785    16.506023

             hum_2        hum_3  windspeed_2  windspeed_3       hour_2  \
count  1000.000000  1000.000000  1000.000000  1000.000000  1000.000000
mean      1.000000    -0.180672     1.000000     0.665403     1.000000
std       1.115265     2.759186     1.769451     6.585134     0.897533
min       0.000002   -39.220902     0.000128    -3.754263     0.002152
25%       0.139237    -0.408761     0.061656    -0.378099     0.152028
50%       0.632432     0.000162     0.491815    -0.000001     0.844552
75%       1.621134     0.616570     1.118505     0.099573     1.593929
max      11.543700     7.004146    27.159723   141.542871     2.885955

            hour_3
count  1000.000000
mean      0.042444
std       1.969596
min      -4.460859
25%      -0.776139
50%      -0.000100
75%       0.563986
max       4.902689

[8 rows x 40 columns]
```

1.5 Write code to add interaction terms to the model. Specifically, we want interactions between the continuous predictors (`temp`,`atemp`, `hum`,`windspeed`) and the month and weekday dummies (`Feb`, `Mar`...`Dec`, `Mon`, `Tue`, ... `Sat`). That means you SHOULD build `atemp*Feb` and `hum*Mon` and so on, but NOT `Feb*Mar` and NOT `Feb*Tue`. The interaction terms should always be a continuous feature times a month dummy or a continuous feature times a weekday dummy.

Encapsulate this process as a function with appropriate inputs and outputs, and test your code by adding interaction terms to `practice_X_train_poly` and show its column names and `.head()`**

```
In [42]: def get_interactions(df, cont_feats = None, dummy_feats = None):
             for c in cont_feats:
                 for d in dummy_feats:
                     df[c + ':'+ d] = df[c]*df[d]
             return df
```

```
        cont_feats = ['temp', 'atemp', 'hum', 'windspeed']
        dummy_feats = ['Feb', 'Mar', 'Apr', 'May', 'Jun', 'Jul', 'Aug', 'Sept', 'Oct', 'Nov',
                       'Mon', 'Tue', 'Wed', 'Thu', 'Fri', 'Sat']

        practice_X_train_poly_ints = get_interactions(practice_X_train_poly, cont_feats, dummy

        display(practice_X_train_poly_ints.columns, practice_X_train_poly_ints.head())
```

```
Index(['hour', 'year', 'holiday', 'temp', 'atemp', 'hum', 'windspeed', 'Feb',
       'Mar', 'Apr',
       ...
       'windspeed:Sept', 'windspeed:Oct', 'windspeed:Nov', 'windspeed:Dec',
       'windspeed:Mon', 'windspeed:Tue', 'windspeed:Wed', 'windspeed:Thu',
       'windspeed:Fri', 'windspeed:Sat'],
      dtype='object', length=108)
```

	hour	year	holiday	temp	atemp	hum	windspeed	\
15762	1.698810	0.982159	0	0.244868	0.248775	0.479363	-0.723106	
4213	-0.046393	-1.018165	0	1.385719	1.132373	-1.538783	0.226495	
14301	-1.355296	0.982159	0	0.867151	0.867002	0.266926	-1.554205	
15900	-0.918995	0.982159	0	-0.999697	-0.988847	0.904236	-0.486103	
14320	1.407943	0.982159	0	1.074578	1.043722	-0.157946	-0.248305	

	Feb	Mar	Apr	...	windspeed:Sept	windspeed:Oct	\
15762	0	0	0	...	-0.0	-0.723106	
4213	0	0	0	...	0.0	0.000000	
14301	0	0	0	...	-0.0	-0.000000	
15900	0	0	0	...	-0.0	-0.486103	
14320	0	0	0	...	-0.0	-0.000000	

	windspeed:Nov	windspeed:Dec	windspeed:Mon	windspeed:Tue	\
15762	-0.0	-0.0	-0.0	-0.723106	
4213	0.0	0.0	0.0	0.000000	
14301	-0.0	-0.0	-0.0	-0.000000	
15900	-0.0	-0.0	-0.0	-0.000000	
14320	-0.0	-0.0	-0.0	-0.000000	

	windspeed:Wed	windspeed:Thu	windspeed:Fri	windspeed:Sat
15762	-0.000000	-0.0	-0.000000	-0.0
4213	0.226495	0.0	0.000000	0.0
14301	-0.000000	-0.0	-1.554205	-0.0
15900	-0.486103	-0.0	-0.000000	-0.0
14320	-0.000000	-0.0	-0.248305	-0.0

[5 rows x 108 columns]

1.6 Combine all your code so far into a function that takes in `bikes_train`, `bikes_val`, the names of columns for polynomial, the target column, the columns to be dropped and produces computation-ready design matrices `X_train` and `X_val` and responses `y_train` and `y_val`. Your final function should build correct, scaled design matrices with the stated interaction terms and any polynomial degree.

```
In [43]: def get_design_mats(train_df, val_df,  degree,
                              columns_forpoly=['temp', 'atemp', 'hum','windspeed', 'hour'],
                              target_col='counts',
                              bad_columns=['counts', 'registered', 'casual', 'workingday', 'mont
             # drop the trivial features in the training & val sets
             practice_X_train = drop_trivials(train_df, bad_columns)
             practice_X_val = drop_trivials(val_df, bad_columns)

             # augment the training & val sets
             practice_X_train_poly = df_augment(practice_X_train, degree, columns_forpoly)
             practice_X_val_poly = df_augment(practice_X_val, degree, columns_forpoly)

             # add interaction terms to training & val sets
             practice_X_train_poly_ints = get_interactions(practice_X_train_poly,
                                                           cont_feats = ['temp', 'atemp', 'hum
                                                          dummy_feats = ['Feb', 'Mar', 'Apr',
                       'Mon', 'Tue', 'Wed', 'Thu', 'Fri', 'Sat'])

             practice_X_val_poly_ints = get_interactions(practice_X_val_poly,
                                                           cont_feats = ['temp', 'atemp', 'hum
                                                          dummy_feats = ['Feb', 'Mar', 'Apr',
                       'Mon', 'Tue', 'Wed', 'Thu', 'Fri', 'Sat'])

             #Standardize training & val sets
             x_train = standardize(practice_X_train_poly_ints, binary_columns = [ 'holiday', 'w
                    'May', 'Jun', 'Jul', 'Aug', 'Sept', 'Oct', 'Nov', 'Dec', 'spring',
                    'summer', 'fall', 'Mon', 'Tue', 'Wed', 'Thu', 'Fri', 'Sat',
                    'Cloudy', 'Snow', 'Storm'])

             x_val = standardize(practice_X_val_poly_ints, binary_columns = [ 'holiday', 'worki
                    'May', 'Jun', 'Jul', 'Aug', 'Sept', 'Oct', 'Nov', 'Dec', 'spring',
                    'summer', 'fall', 'Mon', 'Tue', 'Wed', 'Thu', 'Fri', 'Sat',
                    'Cloudy', 'Snow', 'Storm'])

             #get y_train
             y_train = train_df[target_col]

             #get y_val
             y_val = val_df[target_col]

             return x_train,y_train, x_val,y_val
```

```
x_train, y_train, x_val, y_val = get_design_mats(bikes_train, bikes_val, degree=3)

display(x_train.describe(), y_train.describe(), x_val.describe(), y_val.describe())
```

```
                hour          year      holiday          temp         atemp  \
count  1.000000e+03  1.000000e+03  1000.000000  1.000000e+03  1.000000e+03
mean  -1.994516e-16  2.686740e-17     0.027000  3.019807e-17 -1.256772e-16
std    1.000500e+00  1.000500e+00     0.162164  1.000500e+00  1.000500e+00
min   -1.646163e+00 -1.018165e+00     0.000000 -2.347976e+00 -2.402605e+00
25%   -9.189949e-01 -1.018165e+00     0.000000 -7.922693e-01 -8.121270e-01
50%   -4.639332e-02  9.821591e-01     0.000000  3.744066e-02  7.147176e-02
75%    8.262083e-01  9.821591e-01     0.000000  8.671507e-01  8.670022e-01
max    1.698810e+00  9.821591e-01     1.000000  2.319143e+00  2.546131e+00

                hum     windspeed          Feb          Mar          Apr  \
count  1.000000e+03  1.000000e+03  1000.000000  1000.000000  1000.000000
mean   5.995204e-17  1.301181e-16     0.078000     0.085000     0.082000
std    1.000500e+00  1.000500e+00     0.268306     0.279021     0.274502
min   -3.397602e+00 -1.554205e+00     0.000000     0.000000     0.000000
25%   -7.421467e-01 -7.231056e-01     0.000000     0.000000     0.000000
50%    5.448995e-02 -1.130295e-02     0.000000     0.000000     0.000000
75%    8.511266e-01  4.634972e-01     0.000000     0.000000     0.000000
max    1.913309e+00  5.211499e+00     1.000000     1.000000     1.000000

           ...        windspeed:Sept  windspeed:Oct  windspeed:Nov  \
count      ...          1.000000e+03   1.000000e+03   1.000000e+03
mean       ...          6.972201e-17  -2.806644e-16  -9.386936e-17
std        ...          1.000500e+00   1.000500e+00   1.000500e+00
min        ...         -2.531046e-01  -2.431423e-01  -2.485490e-01
25%        ...         -2.531046e-01  -2.431423e-01  -2.485490e-01
50%        ...         -2.531046e-01  -2.431423e-01  -2.485490e-01
75%        ...         -2.531046e-01  -2.431423e-01  -2.485490e-01
max        ...          9.023019e+00   8.602891e+00   8.203042e+00

       windspeed:Dec  windspeed:Mon  windspeed:Tue  windspeed:Wed  \
count   1.000000e+03   1.000000e+03   1.000000e+03   1.000000e+03
mean   -7.605028e-18   4.751755e-17   2.653433e-17   2.323697e-16
std     1.000500e+00   1.000500e+00   1.000500e+00   1.000500e+00
min    -2.390602e-01  -3.310478e-01  -3.475023e-01  -3.616388e-01
25%    -2.390602e-01  -3.310478e-01  -3.475023e-01  -3.616388e-01
50%    -2.390602e-01  -3.310478e-01  -3.475023e-01  -3.616388e-01
75%    -2.390602e-01  -3.310478e-01  -3.475023e-01  -3.616388e-01
max     9.751170e+00   8.722759e+00   6.313288e+00   5.503679e+00

       windspeed:Thu  windspeed:Fri  windspeed:Sat
count   1.000000e+03   1.000000e+03   1.000000e+03
mean    7.355228e-16  -1.304512e-16  -2.610134e-16
std     1.000500e+00   1.000500e+00   1.000500e+00
```

```
min    -3.235903e-01  -3.094704e-01  -3.417879e-01
25%    -3.235903e-01  -3.094704e-01  -3.417879e-01
50%    -3.235903e-01  -3.094704e-01  -3.417879e-01
75%    -3.235903e-01  -3.094704e-01  -3.417879e-01
max     8.681829e+00   7.251548e+00   6.785282e+00

[8 rows x 108 columns]


count    1000.000000
mean      194.279000
std       191.635042
min         1.000000
25%        35.000000
50%       136.500000
75%       287.250000
max       970.000000
Name: counts, dtype: float64


               hour          year     holiday          temp         atemp  \
count  2.500000e+02  2.500000e+02  250.000000  2.500000e+02  2.500000e+02
mean   1.101341e-16 -5.684342e-17    0.044000  2.646772e-16  1.190159e-16
std    1.002006e+00  1.002006e+00    0.205507  1.002006e+00  1.002006e+00
min   -1.707315e+00 -1.074789e+00    0.000000 -2.199627e+00 -2.421560e+00
25%   -8.374197e-01 -1.074789e+00    0.000000 -8.369150e-01 -8.387994e-01
50%    3.247611e-02  9.304148e-01    0.000000  1.065012e-01  1.287335e-01
75%    9.023719e-01  9.304148e-01    0.000000  8.402694e-01  8.321828e-01
max    1.627285e+00  9.304148e-01    1.000000  2.098158e+00  2.151440e+00

                hum     windspeed        Feb         Mar         Apr  \
count  2.500000e+02  2.500000e+02  250.00000  250.000000  250.000000
mean  -1.381117e-16  9.037215e-17    0.07600    0.084000    0.084000
std    1.002006e+00  1.002006e+00    0.26553    0.277944    0.277944
min   -3.381551e+00 -1.768675e+00    0.00000    0.000000    0.000000
25%   -7.649127e-01 -8.663988e-01    0.00000    0.000000    0.000000
50%   -1.730185e-02 -9.363636e-02    0.00000    0.000000    0.000000
75%    8.237604e-01  6.799895e-01    0.00000    0.000000    0.000000
max    1.958527e+00  3.257303e+00    1.00000    1.000000    1.000000

           ...        windspeed:Sept  windspeed:Oct  windspeed:Nov  \
count      ...          2.500000e+02   2.500000e+02   2.500000e+02
mean       ...          2.087219e-17   1.654232e-16  -2.895462e-16
std        ...          1.002006e+00   1.002006e+00   1.002006e+00
min        ...         -2.698108e-01  -2.462892e-01  -2.418421e-01
25%        ...         -2.698108e-01  -2.462892e-01  -2.418421e-01
50%        ...         -2.698108e-01  -2.462892e-01  -2.418421e-01
75%        ...         -2.698108e-01  -2.462892e-01  -2.418421e-01
```

```
max        ...          5.470366e+00   6.869278e+00   6.200971e+00

       windspeed:Dec  windspeed:Mon  windspeed:Tue  windspeed:Wed  \
count   2.500000e+02   2.500000e+02   2.500000e+02   2.500000e+02
mean   -2.176037e-17  -3.530509e-17   2.504663e-16   2.131628e-17
std     1.002006e+00   1.002006e+00   1.002006e+00   1.002006e+00
min    -2.607248e-01  -3.306931e-01  -3.121535e-01  -3.261101e-01
25%    -2.607248e-01  -3.306931e-01  -3.121535e-01  -3.261101e-01
50%    -2.607248e-01  -3.306931e-01  -3.121535e-01  -3.261101e-01
75%    -2.607248e-01  -3.306931e-01  -3.121535e-01  -3.261101e-01
max     6.890671e+00   5.799685e+00   6.481149e+00   4.856833e+00

       windspeed:Thu  windspeed:Fri  windspeed:Sat
count   2.500000e+02   2.500000e+02   2.500000e+02
mean   -1.723066e-16  -2.735590e-16   1.847411e-16
std     1.002006e+00   1.002006e+00   1.002006e+00
min    -4.102067e-01  -3.466805e-01  -3.405547e-01
25%    -4.102067e-01  -3.466805e-01  -3.405547e-01
50%    -4.102067e-01  -3.466805e-01  -3.405547e-01
75%    -4.102067e-01  -3.466805e-01  -3.405547e-01
max     4.857903e+00   4.708654e+00   6.144642e+00

[8 rows x 108 columns]


count    250.000000
mean     199.576000
std      195.025928
min        1.000000
25%       41.000000
50%      144.000000
75%      291.500000
max      854.000000
Name: counts, dtype: float64
```

Question 2 [20pts]: Regularization via Ridge

2.1 For each degree in 1 through 8:

1. Build the training design matrix and validation design matrix using the function `get_design_mats` with polynomial terms up through the specified degree.
2. Fit a regression model to the training data.
3. Report the model's score on the validation data.

2.2 Discuss patterns you see in the results from 2.1. Which model would you select, and why?

2.3 Let's try regularizing our models via ridge regression. Build a table showing the validation set R^2 of polynomial models with degree from 1-8, regularized at the levels $\lambda =$

$(.01, .05, .1, .5, 1, 5, 10, 50, 100)$. Do not perform cross validation at this point, simply report performance on the single validation set.

2.4 Find the best-scoring degree and regularization combination.

2.5 It's time to see how well our selected model will do on future data. Read in the provided test dataset, do any required formatting, and report the best model's R^2 score. How does it compare to the validation set score that made us choose this model?

2.6 Why do you think our model's test score was quite a bit worse than its validation score? Does the test set simply contain harder examples, or is something else going on?

2.0.2 Solutions

2.1 For each degree in 1 through 8:

1. Build the training design matrix and validation design matrix using the function `get_design_mats` with polynomial terms up through the specified degree.
2. Fit a regression model to the training data.
3. Report the model's score on the validation data.

```
In [44]: higher_order = 8

         for o in range(1,higher_order+1):
             x_train, y_train, x_val, y_val = get_design_mats(bikes_train, bikes_val, degree =
             reg_model = sm.OLS(y_train, sm.add_constant(x_train)).fit()
             pred = reg_model.predict(sm.add_constant(x_val))
             print('The R^2 of degree = '+ str(o) + ' is ' + str(r2_score(y_val, pred)))
```

```
The R^2 of degree = 1 is 0.319599351848
The R^2 of degree = 2 is 0.432296637082
The R^2 of degree = 3 is 0.444765726209
The R^2 of degree = 4 is 0.435727187965
The R^2 of degree = 5 is 0.464701289409
The R^2 of degree = 6 is 0.476456167251
The R^2 of degree = 7 is 0.526835502689
The R^2 of degree = 8 is 0.547186421658
```

2.2 Discuss patterns you see in the results from 2.1. Which model would you select, and why?**

Based on the patterns in 2.1, it seems that the model with polynomial to the degree of 8 is the best model because it has the highest R^2; however, this should raise some red flags, as we may be overfitting the data. Similarly, from degree 1 to 3, the R^2 increases, but then decreases slightly at degree 4. We may also want to think about what the βs look like and how many there are. The R^2 may be increasing, but at what cost to our parameters? These results suggest that we may need regularization to ensure that the model fits a test set in the future.

2.3 Let's try regularizing our models via ridge regression. Build a table showing the validation set R^2 of polynomial models with degree from 1-8, regularized at the levels $\lambda = (.01, .05, .1, .5, 1, 5, 10, 50, 100)$. Do not perform cross validation at this point, simply report performance on the single validation set.

In [45]:
```
#Let's create the dataframe before we fill it
r_squareds = dict() # Store the R^2 from each model in a dictionary

r_squareds['alpha = 0.01'] = [np.nan]*8
r_squareds['alpha = 0.05'] = [np.nan]*8
r_squareds['alpha = 0.1'] = [np.nan]*8
r_squareds['alpha = 0.5'] = [np.nan]*8
r_squareds['alpha = 1'] = [np.nan]*8
r_squareds['alpha = 5'] = [np.nan]*8
r_squareds['alpha = 10'] = [np.nan]*8
r_squareds['alpha = 50'] = [np.nan]*8
r_squareds['alpha = 100'] = [np.nan]*8

dfResults = pd.DataFrame(r_squareds) # Create dataframe

dfResults.rename({0: r'degree 1', 1: r'degree 2', 2: r'degree 3', 3: r'degree 4', 4: r
dfResults
```

Out[45]:

	alpha = 0.01	alpha = 0.05	alpha = 0.1	alpha = 0.5	alpha = 1	\
degree 1	NaN	NaN	NaN	NaN	NaN	
degree 2	NaN	NaN	NaN	NaN	NaN	
degree 3	NaN	NaN	NaN	NaN	NaN	
degree 4	NaN	NaN	NaN	NaN	NaN	
degree 5	NaN	NaN	NaN	NaN	NaN	
degree 6	NaN	NaN	NaN	NaN	NaN	
degree 7	NaN	NaN	NaN	NaN	NaN	
degree 8	NaN	NaN	NaN	NaN	NaN	

	alpha = 10	alpha = 100	alpha = 5	alpha = 50
degree 1	NaN	NaN	NaN	NaN
degree 2	NaN	NaN	NaN	NaN
degree 3	NaN	NaN	NaN	NaN
degree 4	NaN	NaN	NaN	NaN
degree 5	NaN	NaN	NaN	NaN
degree 6	NaN	NaN	NaN	NaN
degree 7	NaN	NaN	NaN	NaN
degree 8	NaN	NaN	NaN	NaN

In [46]:
```
alphas = (.01,.05,.1,.5,1,5,10,50,100)
for a in alphas:
    for o in range(1,higher_order+1):
        x_train, y_train, x_val, y_val = get_design_mats(bikes_train, bikes_val, degre
        clf = Ridge(alpha=a)
        model = clf.fit(x_train, y_train)
        pred = model.predict(x_val)

        #save the R^2s
        new_r2 = r2_score(y_val, pred)
```

```
                dfResults['alpha = ' + str(a)][o-1] = new_r2

        dfResults
```

Out[46]:

	alpha = 0.01	alpha = 0.05	alpha = 0.1	alpha = 0.5	alpha = 1	\
degree 1	0.320453	0.322941	0.325128	0.333028	0.336420	
degree 2	0.432869	0.434304	0.435476	0.439646	0.441552	
degree 3	0.446378	0.450655	0.454360	0.465101	0.467141	
degree 4	0.441839	0.446990	0.450112	0.460595	0.465102	
degree 5	0.443535	0.444726	0.447795	0.457971	0.462325	
degree 6	0.447306	0.444154	0.446298	0.456232	0.460549	
degree 7	0.447461	0.443823	0.445193	0.454678	0.459101	
degree 8	0.450000	0.443482	0.444360	0.453210	0.457754	

	alpha = 10	alpha = 100	alpha = 5	alpha = 50
degree 1	0.338836	0.333825	0.339523	0.335637
degree 2	0.434478	0.368081	0.440610	0.392415
degree 3	0.457921	0.405528	0.462171	0.431134
degree 4	0.466735	0.430218	0.468587	0.450390
degree 5	0.467927	0.443860	0.467920	0.458462
degree 6	0.466773	0.450138	0.466249	0.460721
degree 7	0.465389	0.452197	0.464842	0.460413
degree 8	0.464222	0.452127	0.463704	0.459227

2.4 Find the best-scoring degree and regularization combination.

```
In [47]: max_result = dfResults.max().max()
         dfResults[dfResults == max_result].stack().index.tolist()

         print('The maximum R^2: \n ' + str(max_result) +'\nDegree and alpha for that R^2: \n'
```

```
The maximum R^2:
 0.468586510367
Degree and alpha for that R^2:
[('degree 4', 'alpha = 5')]
```

2.5 It's time to see how well our selected model will do on future data. Read in the provided test dataset `data/bikes_test.csv`, do any required formatting, and report the best model's R^2 score. How does it compare to the validation set score that made us choose this model?

```
In [48]: bikes_test = pd.read_csv('data/bikes_test.csv', sep=",", index_col = 'Unnamed: 0')
         bikes_test.head()
         bikes_test.shape
```

Out[48]: (1250, 36)

```
In [49]: #Do required formatting
         def get_design_test(train_df, test_df,  degree,
```

```
                                columns_forpoly=['temp', 'atemp', 'hum','windspeed', 'hour'],
                                target_col='counts',
                                bad_columns=['counts', 'registered', 'casual', 'workingday', 'mont
            # drop the trivial features in the training & val sets
            practice_X_train = drop_trivials(train_df, bad_columns)
            practice_X_test = drop_trivials(test_df, bad_columns)

            # augment the training & val sets
            practice_X_train_poly = df_augment(practice_X_train, degree, columns_forpoly)
            practice_X_test_poly = df_augment(practice_X_test, degree, columns_forpoly)

            # add interaction terms to training & val sets
            practice_X_train_poly_ints = get_interactions(practice_X_train_poly,
                                                          cont_feats = ['temp', 'atemp', 'hum
                                                         dummy_feats = ['Feb', 'Mar', 'Apr',
                       'Mon', 'Tue', 'Wed', 'Thu', 'Fri', 'Sat'])

            practice_X_test_poly_ints = get_interactions(practice_X_test_poly,
                                                          cont_feats = ['temp', 'atemp', 'hum
                                                         dummy_feats = ['Feb', 'Mar', 'Apr',
                       'Mon', 'Tue', 'Wed', 'Thu', 'Fri', 'Sat'])

            #Standardize training & val sets
            x_train = standardize(practice_X_train_poly_ints, binary_columns = [ 'holiday', 'w
                   'May', 'Jun', 'Jul', 'Aug', 'Sept', 'Oct', 'Nov', 'Dec', 'spring',
                   'summer', 'fall', 'Mon', 'Tue', 'Wed', 'Thu', 'Fri', 'Sat',
                   'Cloudy', 'Snow', 'Storm'])

            x_test = standardize(practice_X_test_poly_ints, binary_columns = [ 'holiday', 'wo
                   'May', 'Jun', 'Jul', 'Aug', 'Sept', 'Oct', 'Nov', 'Dec', 'spring',
                   'summer', 'fall', 'Mon', 'Tue', 'Wed', 'Thu', 'Fri', 'Sat',
                   'Cloudy', 'Snow', 'Storm'])

            #get y_train
            y_train = train_df[target_col]

            #get y_val
            y_test = test_df[target_col]

            return x_train, y_train, x_test, y_test

        x_train, y_train, x_test, y_test = get_design_test(bikes_main, bikes_test, degree=4)

        display(x_train.describe(), y_train.describe(), x_test.describe(), y_test.describe())
```

```
              hour          year      holiday          temp         atemp  \
count  1.250000e+03  1.250000e+03  1250.000000  1.250000e+03  1.250000e+03
mean   1.216804e-16  8.100187e-17     0.030400 -2.156497e-16 -1.323386e-17
```

```
std    1.000400e+00  1.000400e+00     0.171754  1.000400e+00  1.000400e+00
min   -1.657838e+00 -1.029227e+00     0.000000 -2.359867e+00 -2.406224e+00
25%   -9.313786e-01 -1.029227e+00     0.000000 -8.010330e-01 -8.174189e-01
50%   -5.962775e-02  9.716030e-01     0.000000  3.034531e-02  6.525066e-02
75%    8.121230e-01  9.716030e-01     0.000000  8.617237e-01  8.599446e-01
max    1.683874e+00  9.716030e-01     1.000000  2.316636e+00  2.537308e+00

                hum     windspeed          Feb          Mar          Apr  \
count  1.250000e+03  1.250000e+03  1250.000000  1250.000000  1250.000000
mean  -3.493872e-17 -2.641443e-16     0.077600     0.084800     0.082400
std    1.000400e+00  1.000400e+00     0.267648     0.278695     0.275083
min   -3.394074e+00 -1.592792e+00     0.000000     0.000000     0.000000
25%   -7.359745e-01 -7.491944e-01     0.000000     0.000000     0.000000
50%    6.145525e-02 -2.668770e-02     0.000000     0.000000     0.000000
75%    8.588850e-01  6.966263e-01     0.000000     0.000000     0.000000
max    1.922125e+00  5.274655e+00     1.000000     1.000000     1.000000

           ...        windspeed:Sept  windspeed:Oct  windspeed:Nov  \
count      ...          1.250000e+03   1.250000e+03   1.250000e+03
mean       ...         -8.819612e-17   2.800427e-16   1.976641e-16
std        ...          1.000400e+00   1.000400e+00   1.000400e+00
min        ...         -2.564578e-01  -2.436104e-01  -2.471551e-01
25%        ...         -2.564578e-01  -2.436104e-01  -2.471551e-01
50%        ...         -2.564578e-01  -2.436104e-01  -2.471551e-01
75%        ...         -2.564578e-01  -2.436104e-01  -2.471551e-01
max        ...          8.813854e+00   8.735199e+00   8.333669e+00

       windspeed:Dec  windspeed:Mon  windspeed:Tue  windspeed:Wed  \
count   1.250000e+03   1.250000e+03   1.250000e+03   1.250000e+03
mean   -8.731238e-16   1.524114e-16  -1.256772e-16   7.016610e-17
std     1.000400e+00   1.000400e+00   1.000400e+00   1.000400e+00
min    -2.432744e-01  -3.308045e-01  -3.396932e-01  -3.544707e-01
25%    -2.432744e-01  -3.308045e-01  -3.396932e-01  -3.544707e-01
50%    -2.432744e-01  -3.308045e-01  -3.396932e-01  -3.544707e-01
75%    -2.432744e-01  -3.308045e-01  -3.396932e-01  -3.544707e-01
max     9.346317e+00   8.581266e+00   6.942585e+00   5.686829e+00

       windspeed:Thu  windspeed:Fri  windspeed:Sat
count   1.250000e+03   1.250000e+03   1.250000e+03
mean    1.281641e-16   4.121148e-17   4.984457e-16
std     1.000400e+00   1.000400e+00   1.000400e+00
min    -3.418231e-01  -3.169780e-01  -3.415251e-01
25%    -3.418231e-01  -3.169780e-01  -3.415251e-01
50%    -3.418231e-01  -3.169780e-01  -3.415251e-01
75%    -3.418231e-01  -3.169780e-01  -3.415251e-01
max     8.341481e+00   7.210258e+00   6.822161e+00

[8 rows x 113 columns]
```

```
count    1250.000000
mean      195.338400
std       192.251045
min         1.000000
25%        36.000000
50%       137.000000
75%       289.500000
max       970.000000
Name: counts, dtype: float64
```

```
                hour          year     holiday          temp         atemp  \
count   1.250000e+03  1.250000e+03  1250.00000  1.250000e+03  1.250000e+03
mean    9.592327e-18 -1.993072e-16     0.02400  1.030287e-17  1.803002e-16
std     1.000400e+00  1.000400e+00     0.15311  1.000400e+00  1.000400e+00
min    -1.639854e+00 -9.747194e-01     0.00000 -2.337574e+00 -2.573327e+00
25%    -7.805717e-01 -9.747194e-01     0.00000 -8.040013e-01 -8.206155e-01
50%     7.871023e-02 -9.747194e-01     0.00000  1.390440e-02  5.574015e-02
75%     7.947785e-01  1.025936e+00     0.00000  8.318101e-01  8.447495e-01
max     1.654061e+00  1.025936e+00     1.00000  2.365383e+00  2.334843e+00

                hum     windspeed          Feb          Mar          Apr  \
count   1.250000e+03  1.250000e+03  1250.000000  1250.000000  1250.000000
mean   -2.398082e-17 -4.121148e-17     0.077600     0.084800     0.082400
std     1.000400e+00  1.000400e+00     0.267648     0.278695     0.275083
min    -3.249534e+00 -1.549287e+00     0.000000     0.000000     0.000000
25%    -7.410163e-01 -7.025532e-01     0.000000     0.000000     0.000000
50%    -9.365134e-03 -2.188212e-01     0.000000     0.000000     0.000000
75%     8.268076e-01  5.063716e-01     0.000000     0.000000     0.000000
max     1.976545e+00  4.497768e+00     1.000000     1.000000     1.000000

           ...        windspeed:Sept  windspeed:Oct  windspeed:Nov  \
count      ...          1.250000e+03   1.250000e+03   1.250000e+03
mean       ...         -4.305001e-16   2.003731e-16  -3.399947e-16
std        ...          1.000400e+00   1.000400e+00   1.000400e+00
min        ...         -2.531663e-01  -2.449452e-01  -2.493925e-01
25%        ...         -2.531663e-01  -2.449452e-01  -2.493925e-01
50%        ...         -2.531663e-01  -2.449452e-01  -2.493925e-01
75%        ...         -2.531663e-01  -2.449452e-01  -2.493925e-01
max        ...          7.041538e+00   8.202304e+00   7.818134e+00

       windspeed:Dec  windspeed:Mon  windspeed:Tue  windspeed:Wed  \
count   1.250000e+03   1.250000e+03   1.250000e+03   1.250000e+03
mean   -2.131628e-18  -2.978062e-16  -1.912248e-16   1.835865e-16
std     1.000400e+00   1.000400e+00   1.000400e+00   1.000400e+00
min    -2.525008e-01  -3.473844e-01  -3.528308e-01  -3.530606e-01
```

```
25%    -2.525008e-01  -3.473844e-01  -3.528308e-01  -3.530606e-01
50%    -2.525008e-01  -3.473844e-01  -3.528308e-01  -3.530606e-01
75%    -2.525008e-01  -3.473844e-01  -3.528308e-01  -3.530606e-01
max     8.238439e+00   7.620106e+00   5.753353e+00   6.354934e+00

       windspeed:Thu  windspeed:Fri  windspeed:Sat
count   1.250000e+03   1.250000e+03   1.250000e+03
mean   -6.657785e-16  -6.930456e-16  -9.192647e-17
std     1.000400e+00   1.000400e+00   1.000400e+00
min    -3.262144e-01  -3.439132e-01  -3.100820e-01
25%    -3.262144e-01  -3.439132e-01  -3.100820e-01
50%    -3.262144e-01  -3.439132e-01  -3.100820e-01
75%    -3.262144e-01  -3.439132e-01  -3.100820e-01
max     5.825396e+00   9.196647e+00   7.853491e+00

[8 rows x 113 columns]


count    1250.000000
mean      193.318400
std       180.892555
min         1.000000
25%        46.000000
50%       151.000000
75%       277.000000
max       967.000000
Name: counts, dtype: float64
```

```
In [50]: #Fit the best model to the test data

         clf = Ridge(alpha=5)
         model = clf.fit(x_train, y_train)
         pred = model.predict(x_test)

         #Report the R2

         test_r2 = r2_score(y_test, pred)
         test_r2
```

```
Out[50]: 0.51665756599214285
```

2.6 Why do you think our model's test score was quite a bit worse than its validation score? Does the test set simply contain harder examples, or is something else going on?

We didn't use cross-validation, the data could have not been shuffled correctly, or there may be bias in the train/val sets.

Question 3 [20pts]: Comparing Ridge, Lasso, and OLS

3.1 Build a dataset with polynomial degree 1 and fit an OLS model, a Ridge model, and a Lasso model. Use `RidgeCV` and `LassoCV` to select the best regularization level from among `(.1,.5,1,5,10,50,100)`.

Note: On the lasso model, you will need to increase `max_iter` to 100,000 for the optimization to converge.

3.2 Plot histograms of the coefficients found by each of OLS, ridge, and lasso. What trends do you see in the magnitude of the coefficients?

3.3 The plots above show the overall distribution of coefficient values in each model, but do not show how each model treats individual coefficients. Build a plot which cleanly presents, for each feature in the data, 1) The coefficient assigned by OLS, 2) the coefficient assigned by ridge, and 3) the coefficient assigned by lasso.

Hint: Bar plots are a possible choice, but you are not required to use them

Hint: use `xticks` to label coefficients with their feature names

3.4 What trends do you see in the plot above? How do the three approaches handle the correlated pair `temp` and `atemp`?

2.0.3 Solutions

3.1 Build a dataset with polynomial degree 1 and fit an OLS model, a Ridge model, and a Lasso model. Use `RidgeCV` and `LassoCV` to select the best regularization level from among `(.1,.5,1,5,10,50,100)`.

Note: On the lasso model, you will need to increase `max_iter` to 100,000 for the optimization to converge.

```
In [53]: x_train, y_train, x_test, y_test = get_design_test(bikes_main, bikes_test, degree=1)
```

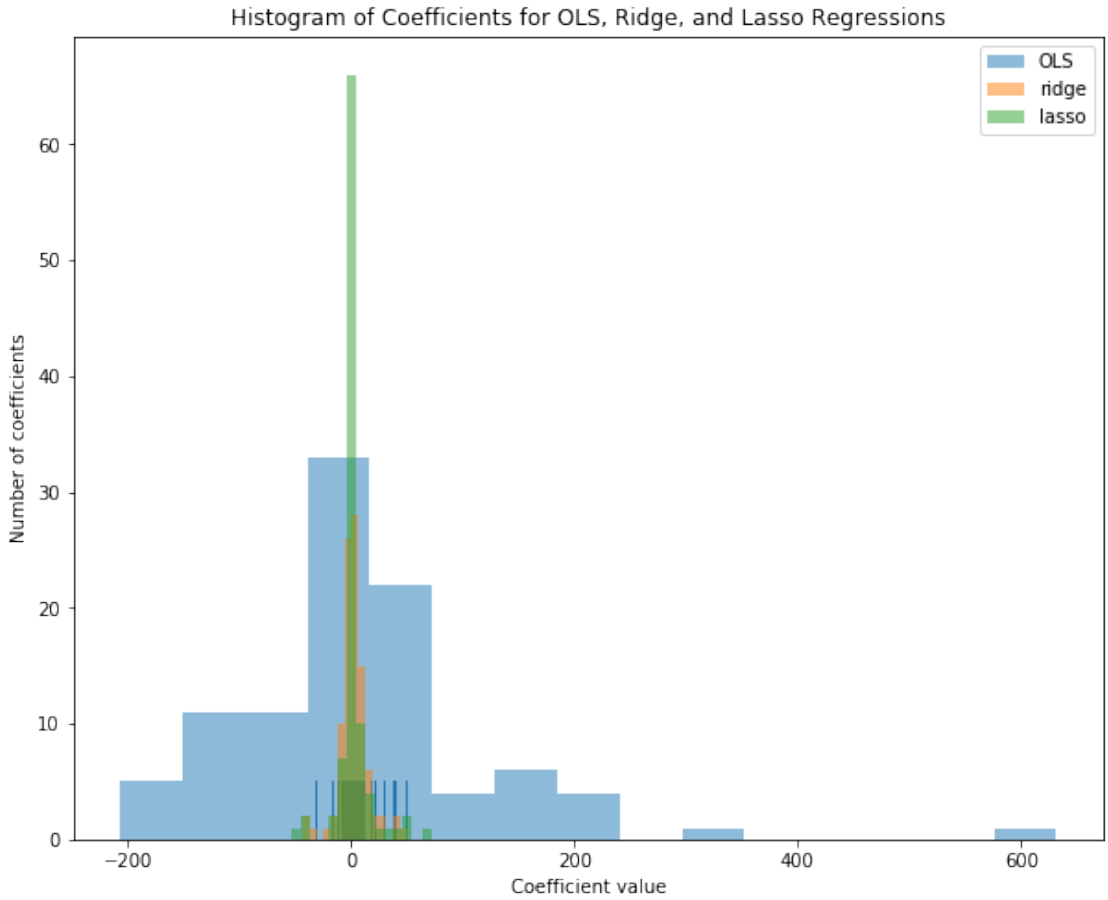
```
In [56]: x_train, y_train, x_val, y_val = get_design_mats(bikes_train, bikes_val, 1)

         alpha = (.01, .05, .1,.5, 1, 5, 10, 50, 100)

         OLSmodel = sm.OLS(y_train, sm.add_constant(x_train)).fit()
         OLSpredict = OLSmodel.predict(sm.add_constant(x_val))

         ridge = RidgeCV(alphas = alpha, store_cv_values=True)
         ridge_model = ridge.fit(x_train, y_train)
         ypredict_ridge = ridge.predict(x_val)
         print("Best alpha using built-in RidgeCV: %f" % ridge_model.alpha_)

         lasso = LassoCV(alphas = alpha, max_iter=100000, cv=None)
         lasso_model = lasso.fit(x_train, y_train)
         print('Best alpha using built-in LassoCV: %f' % lasso_model.alpha_)
```

```
Best alpha using built-in RidgeCV: 50.000000
Best alpha using built-in LassoCV: 1.000000
```

3.2 Plot histograms of the coefficients found by each of OLS, ridge, and lasso. What trends do you see in the magnitude of the coefficients?

```
In [59]: ols_coef_list = OLSmodel.params
         ols_coef= np.asarray(ols_coef_list[1:99])
         ridge_coef_50 = (Ridge(alpha=50).fit(x_train,y_train).coef_)
         lasso_coef_1 = (Lasso(alpha=1).fit(x_train,y_train).coef_)
```

```
In [65]: fig, ax = plt.subplots(figsize=(10, 8))
         plt.bar(ridge_coef_50,5, width=0.8)
         ols_hist = plt.hist(ols_coef, alpha =.5, bins = 15, label = 'OLS');
         ridge_hist = plt.hist(ridge_coef_50, alpha=.5, bins = 15, label = 'ridge');
         lasso_hist = plt.hist(lasso_coef_1, alpha=.5, bins = 15, label = 'lasso');
         plt.title('Histogram of Coefficients for OLS, Ridge, and Lasso Regressions');
         plt.legend();
         plt.xlabel('Coefficient value');
         plt.ylabel('Number of coefficients');
```



We see on this graph that the majority of coefficients have a magnitude close to zero in all three models.

3.3 The plots above show the overall distribution of coefficient values in each model, but do not show how each model treats individual coefficients. Build a plot which cleanly presents, for each feature in the data, 1) The coefficient assigned by OLS, 2) the coefficient assigned by ridge, and 3) the coefficient assigned by lasso.

Hint: Bar plots are a possible choice, but you are not required to use them

Hint: use `xticks` to label coefficients with their feature names

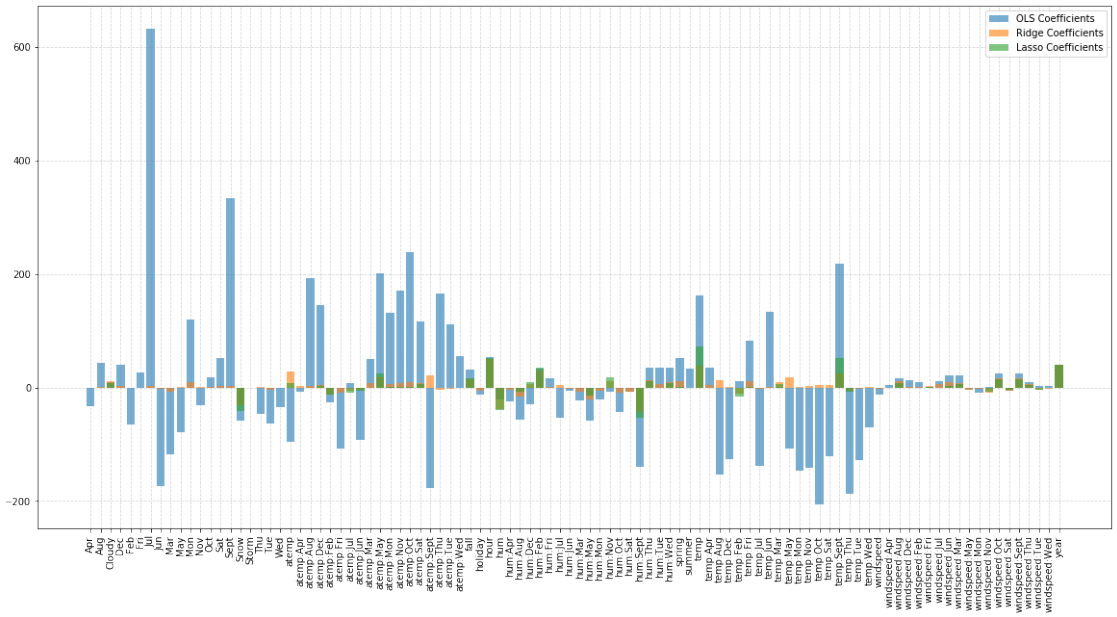
```
In [66]: ols_coef_df = pd.DataFrame(ols_coef, index = x_train.columns.values)
         ridge_coef_df = pd.DataFrame(ridge_coef_50, index = x_train.columns.values)
```

```
        lasso_coef_df = pd.DataFrame(lasso_coef_1, index = x_train.columns.values)
        coef_df = pd.concat([ols_coef_df, ridge_coef_df, lasso_coef_df],axis = 1,join_axes=[ol
        coef_df.columns = ['OLS', 'Ridge','Lasso']
        display(coef_df.loc['temp'], coef_df.loc['atemp'])
```

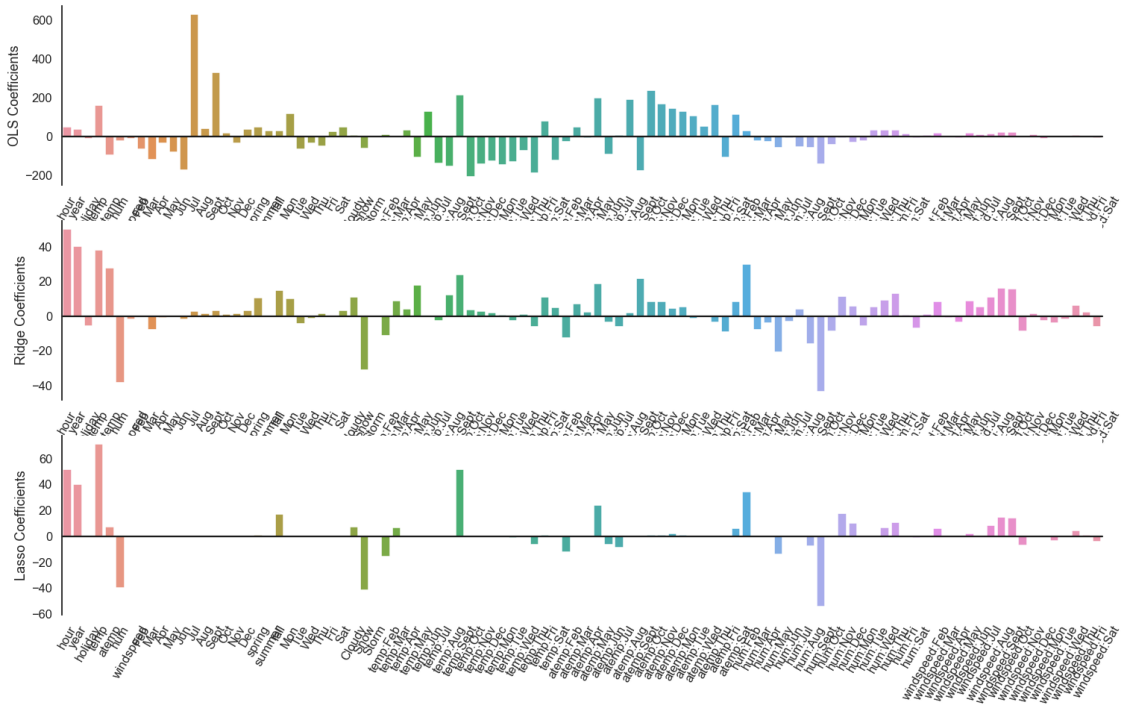
```
OLS      162.090583
Ridge     38.037967
Lasso     71.547049
Name: temp, dtype: float64
```

```
OLS     -95.858548
Ridge    27.661118
Lasso     7.133555
Name: atemp, dtype: float64
```

```
In [68]: # Build Bar Plot
        x_vals = x_train.columns.values
        plt.figure(figsize=(20,10));
        plt.grid(b=True, which='major', color='tab:gray', alpha = .3, linestyle='--');
        plt.xticks(rotation=90);
        ols_bar = plt.bar(x_vals, ols_coef, alpha = .6);
        ridge_bar = plt.bar(x_vals, ridge_coef_50, alpha =.6);
        lasso_bar =plt.bar(x_vals, lasso_coef_1, alpha = .6);
        plt.legend([ols_bar, ridge_bar, lasso_bar], ['OLS Coefficients','Ridge Coefficients',
```



In [71]:
```
import seaborn as sns

sns.set(style="white", context="talk")
rs = np.random.RandomState(8)

# Set up the matplotlib figure
f, (ax1, ax2, ax3) = plt.subplots(3, 1, figsize=(25, 15))

# OLS
x = coef_df.index
y1 = coef_df['OLS']
sns.barplot(x=x, y=y1, ax=ax1)
ax1.axhline(0, color="k", clip_on=False)
ax1.set_ylabel("OLS Coefficients")
ax1.set_xticklabels(coef_df.index, rotation=60)

# Ridge
y2 = coef_df['Ridge']
sns.barplot(x=x, y=y2, ax=ax2)
ax2.axhline(0, color="k", clip_on=False)
ax2.set_ylabel("Ridge Coefficients")
ax2.set_xticklabels(coef_df.index, rotation=60)

# Lasso
y3 = coef_df['Lasso']
sns.barplot(x=x, y=y3, ax=ax3)
ax3.axhline(0, color="k", clip_on=False)
ax3.set_ylabel("Lasso Coefficients")
ax3.set_xticklabels(coef_df.index,rotation=60)

# Finalize the plot
sns.despine(bottom=True)
```



3.4 What trends do you see in the plot above? How do the three approaches handle the correlated pair `temp` and `atemp`?

The Ridge and Lasso regressions have smaller coefficients than the OLS model, because OLS is more sensitive to a change in any feature. In terms of penalization, The Lasso regression was most sensitive to penalization and left us with about 8 main coefficients. The Ridge regression supplied a greater number of significant coefficients than Lasso, but at less extreme values than OLS.

In terms of `temp` and `atemp`, each model handled the pair differently. OLS chose two large values with inverse signs (162 and -95, respectively). Generally speaking, the `atemp` coefficient will negate the `temp` coefficient so their combined effective is net positive. Ridge and Lasso both returned positive coefficients for `temp` and `atemp`. Both gave higher weights to `temp`, which is consistent with the OLS coefficient selection.

Question 4 [20 pts]: Reflection

These problems are open-ended, and you are not expected to write more than 2-3 sentences. We are interested in seeing that you have thought about these issues; you will be graded on how well you justify your conclusions here, not on what you conclude.

4.1 Reflect back on the `get_design_mats` function you built. Writing this function useful in your analysis? What issues might you have encountered if you copy/pasted the model-building code instead of tying it together in a function? Does a `get_design_mat` function seem wise in general, or are there better options?

Writing this function was extremely useful in my analysis, because it streamlined the process of creating datasets with the desired features for each part of the assignment. Had I tried to copy/paste the model-building code, I would've definitely run into human error along the way, as even in building the function I ran into some errors that needed to be corrected. I think it's wise to build a function like this. There may be other ways to implement the function (with built in python packages, for example), but it's nice to have control over your code in this way.

4.2 What are the costs and benefits of applying ridge/lasso regularization to an overfit OLS model, versus setting a specific degree of polynomial or forward selecting features for the model?

Ridge and Lasso minimize the impact of irrelevant features on our trained model. While Ridge only minimizes, Lasso can set them to zero. This can help highlight the most relevant features, and make the model more interpretable. The overfit OLS is difficult to interpret. Ridge is preferred when you know the parameters you want and want to add larger penalties to larger coefficients. LASSO is preferred when you want feature selection.

Forward selection procedure was cumbersome to code, and took longer to iterate, and used more computational energy than applying ridge/lasso.

** 4.3** This pset posed a purely predictive goal: forecast ridership as accurately as possible. How important is interpretability in this context? Considering, e.g., your lasso and ridge models from Question 3, how would you react if the models predicted well, but the coefficient values didn't make sense once interpreted?

In a context where forecasting is most important, interpretability is not as important as accuracy. You want to be able to plug in the knowns for the predictors and get as accurate as possible of an outcome. In the world of big data, this might mean having 100s of features, which makes the model very uninterpritable. If interpretability was of higher importance, we would want to select a parsimonious model that told "enough" of the story that it could be easily interpreted without losing too much acuracy. It really just depends on your goals.

4.4 Reflect back on our original goal of helping BikeShare predict what demand will be like in the week ahead, and thus how many bikes they can bring in for maintenance. In your view, did we accomplish this goal? If yes, which model would you put into production and why? If not, which model came closest, what other analyses might you conduct, and how likely do you think they are to work

I think we were successful in building a model for this goal. Given this, I would use the Ridge model because it seems to be the most parsimonous while also not losing key features that would help with prediction. The Ridge and Lasso models both show that the same key features are important. However, the Lasso model takes out many features that might help with more accurate predictions.