CS109A Introduction to Data Science:

Homework 4 - Regularization

Harvard University
Fall 2018
Instructors: Pavlos Protopapas, Kevin Rader

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:

Type *Markdown* and LaTeX: α^2

In [1]:
```
#RUN THIS CELL
import requests
from IPython.core.display import HTML
styles = requests.get("https://raw.githubusercontent.com/Harvard-IACS/2018-CS109A/
HTML(styles)
```

Out[1]:

import these libraries

In [2]:
```
import warnings
#warnings.filterwarnings('ignore')
import numpy as np
import pandas as pd
import matplotlib
import matplotlib.pyplot as plt
import seaborn as sns
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.metrics import mean_squared_error
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
```

```
C:\Users\winsl0w\Anaconda3\lib\site-packages\ipykernel_launcher.py:26: FutureWa
rning: The pandas.core.datetools module is deprecated and will be removed in a
future version. Please use the pandas.tseries module instead.
```

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 (https://www.capitalbikeshare.com) 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.

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 [3]:
```
# read in and split bikes dataframe
def initialize_and_split_bikes(path: str, test_size=0.2):

    # intialize df from csv, specifying Unnamed:0 as the index
    df = pd.read_csv(path, index_col="Unnamed: 0")

    # split data into train and test, stratify by month
    train_df, test_df = train_test_split(df, test_size=test_size, random_state=90
    return(train_df, test_df)
```

In [4]:
```
# read in dfs and print header rows
bikes_train, bikes_val = initialize_and_split_bikes("data/bikes_student.csv")
bikes_train.head()
```

Out[4]:

	dteday	hour	year	holiday	workingday	temp	atemp	hum	windspeed	casual	...	Mon
15762	2012-10-23	23	1	0	1	0.54	0.5152	0.73	0.1045	9	...	0
4213	2011-06-29	11	0	0	1	0.76	0.6667	0.35	0.2239	53	...	0
14301	2012-08-24	2	1	0	1	0.66	0.6212	0.69	0.0000	1	...	0
15900	2012-10-31	5	1	0	1	0.30	0.3030	0.81	0.1343	0	...	0
14320	2012-08-24	21	1	0	1	0.70	0.6515	0.61	0.1642	58	...	0

5 rows × 36 columns

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 [5]:
```
# split input df into predictors and response
def select_predictors_response(df, drop_cols, response_col):

    # drop unwanted columns
    df = df.drop(drop_cols, axis=1)

    # output all remaining columns minus the response, and the response separatel
    return(df.drop(response_col, axis=1),df[response_col])
```

In [7]:
```
# define columns to drop
drop_cols = ["registered","casual","workingday","month","dteday"]

# drop columns and split dfs into response and predictor
practice_x_train, practice_y_train = select_predictors_response(bikes_train, drop
practice_x_val, practice_y_val = select_predictors_response(bikes_val, drop_cols,
```

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 [8]:
```
# standardize specified columns of df with the scaling params specified by scaler
def standardize_cols(df, cols, scaler=None):

    # if no scaler is input, fit scaler to the input data
    if scaler is None:
        scaler = StandardScaler().fit(df[cols].values)

    # transform and output scaled data scaler
    df[cols] = scaler.transform(df[cols].values)
    return(df, scaler)
```

In [14]:
```
# get columns to standardize
binary_columns = [ 'holiday','Feb', 'Mar', 'Apr',
       'May', 'Jun', 'Jul', 'Aug', 'Sept', 'Oct', 'Nov', 'Dec', 'spring',
       'summer', 'fall', 'Mon', 'Tue', 'Wed', 'Thu', 'Fri', 'Sat',
       'Cloudy', 'Snow', 'Storm']
nonbinary_cols = practice_x_train.columns.difference(binary_columns)

# standardize columns of training data and save scaler for transforming validatio
practice_x_train_scaled, x_train_scaler = standardize_cols(practice_x_train, nonb
```

In [15]:
```
practice_x_train_scaled.describe()
```

Out[15]:

	hour	year	holiday	temp	atemp	hum	
count	1.000000e+03	1.000000e+03	1000.000000	1.000000e+03	1.000000e+03	1.000000e+03	1.(
mean	4.114764e-17	2.375877e-17	0.027000	3.330669e-18	-3.530509e-17	1.842970e-17	3.
std	1.000500e+00	1.000500e+00	0.162164	1.000500e+00	1.000500e+00	1.000500e+00	1.(
min	-1.646163e+00	-1.018165e+00	0.000000	-2.347976e+00	-2.402605e+00	-3.397602e+00	-1.!
25%	-9.189949e-01	-1.018165e+00	0.000000	-7.922693e-01	-8.121270e-01	-7.421467e-01	-7.
50%	-4.639332e-02	9.821591e-01	0.000000	3.744066e-02	7.147176e-02	5.448995e-02	-1.
75%	8.262083e-01	9.821591e-01	0.000000	8.671507e-01	8.670022e-01	8.511266e-01	4.
max	1.698810e+00	9.821591e-01	1.000000	2.319143e+00	2.546131e+00	1.913309e+00	5.:

8 rows × 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 [11]:
```
# add pure powers of cols columns in df up to the degree-th power
def add_poly_terms(df, cols, degree=2):

    # do nothing degree is negative, zero, or one
    if degree < 2:
        return(df)

    # for each column separately
    for col in cols:

        # create str for new col name
        new_col_names = [col + "_" + str(i) for i in range(2,degree+1)]

        # initialize placeholder data frame with new columns up to the degree-th
        tmp_df = pd.DataFrame(columns=new_col_names, \
                    data=np.array(list(map(lambda x: np.power(df[col].values,x),\
                    np.arange(2,degree+1)))).T, index=df.index)

        # append placeholder dataframe to df
        df = pd.concat((df,tmp_df), axis=1)

    return(df)
```

In [13]:
```
# define columns to raise to degree power
poly_cols = ["temp","atemp","hum","windspeed","hour"]

# create new df with added terms
practice_x_train_poly = add_poly_terms(practice_x_train_scaled, poly_cols, degree

# print column names and header rows
print(practice_x_train_poly.columns)
practice_x_train_poly.head()
```

```
Index(['hour', 'year', 'holiday', 'temp', 'atemp', 'hum', 'windspeed', 'Feb',
       'Mar', 'Apr', 'May', 'Jun', 'Jul', 'Aug', 'Sept', 'Oct', 'Nov', 'Dec',
       'spring', 'summer', 'fall', 'Mon', 'Tue', 'Wed', 'Thu', 'Fri', 'Sat',
       'Cloudy', 'Snow', 'Storm', 'temp_2', 'temp_3', 'temp_4', 'atemp_2',
       'atemp_3', 'atemp_4', 'hum_2', 'hum_3', 'hum_4', 'windspeed_2',
       'windspeed_3', 'windspeed_4', 'hour_2', 'hour_3', 'hour_4'],
      dtype='object')
```

Out[13]:

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

5 rows × 45 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 [16]:
```
# add new interactor terms to df from all pairwise combinations of the
# specified continuous (cont) and dummy columns
def add_interactor_terms(df, cont_cols, dummy_cols):

    # get numeric values of dummy columns
    x = df[dummy_cols].values

    # for each continuous predictor in cont_cols
    for col in cont_cols:

        # create new str for interactor column name
        new_col_names = [col + "_x_" + d for d in dummy_cols]

        # broadcast multiply col values with dummy colums, store in tmp dataframe
        tmp_df = pd.DataFrame(data=df[col].values.reshape(-1,1)*x, columns=new_co

        # append new columns to existing df
        df = pd.concat((df,tmp_df), axis=1)

    return(df)
```

In [17]:
```
# define dummy and continuous predictor column names
dummies = ["Feb", "Mar", "Apr", "May", "Jun", "Jul", "Aug", "Sept", "Oct",\
           "Nov", "Dec", "Mon", "Tue", "Wed", "Thu", "Fri", "Sat"]
continuous =["temp","atemp","hum","windspeed"]

# add new terms and display header rows
practice_x_train_poly = add_interactor_terms(practice_x_train_poly, continuous, d
practice_x_train_poly.head()
```

Out[17]:

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

5 rows × 113 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 [18]:
```
def get_design_mats(train_df, val_df,  degree,
                    columns_forpoly=['temp', 'atemp', 'hum','windspeed', 'hour'],
                    target_col='counts',
                    bad_columns=['registered', 'casual', 'workingday', 'month', '
                    add_interactors = True,
                   ):

    # drop bad columns and split response var from predictors
    x_train, y_train = select_predictors_response(train_df, bad_columns, target_c
    x_val, y_val = select_predictors_response(val_df, bad_columns, target_col)

    # standardize continuous columns (defined here as columns of float dtype rath
    nb_cols = x_train.select_dtypes(include=["float"]).columns
    x_train, x_train_scaler = standardize_cols(x_train, nb_cols)
    x_val, x_train_scaler = standardize_cols(x_val, nb_cols, scaler=x_train_scale

    # add polynomial terms
    x_train = add_poly_terms(x_train, columns_forpoly, degree=degree)
    x_val = add_poly_terms(x_val, columns_forpoly, degree=degree)

    # add interactor terms (optional)
    if add_interactors:
        dummies = ["Feb","Mar","Apr","May","Jun","Jul","Aug","Sept","Oct","Nov",
                   "Dec","Mon","Tue","Wed","Thu","Fri","Sat"]
        continuous = ["temp","atemp","hum","windspeed"]
        x_train = add_interactor_terms(x_train, continuous, dummies)
        x_val = add_interactor_terms(x_val, continuous, dummies)

    # output data
    return(x_train, y_train, x_val, y_val)
```

In [19]:
```
# import data and pre-process
bikes_train, bikes_val = initialize_and_split_bikes("data/bikes_student.csv")

# get standardized design matrices with interactor and polynomial terms
x_train, y_train, x_val, y_val = get_design_mats(bikes_train, bikes_val,  2)
x_train.describe()
```

Out[19]:

	hour	year	holiday	temp	atemp	hum	win
count	1000.000000	1000.000000	1000.000000	1.000000e+03	1.000000e+03	1.000000e+03	1.000(
mean	11.319000	0.509000	0.027000	3.019807e-17	-1.256772e-16	5.995204e-17	1.301
std	6.879431	0.500169	0.162164	1.000500e+00	1.000500e+00	1.000500e+00	1.000[
min	0.000000	0.000000	0.000000	-2.347976e+00	-2.402605e+00	-3.397602e+00	-1.554[
25%	5.000000	0.000000	0.000000	-7.922693e-01	-8.121270e-01	-7.421467e-01	-7.231
50%	11.000000	1.000000	0.000000	3.744066e-02	7.147176e-02	5.448995e-02	-1.130
75%	17.000000	1.000000	0.000000	8.671507e-01	8.670022e-01	8.511266e-01	4.634
max	23.000000	1.000000	1.000000	2.319143e+00	2.546131e+00	1.913309e+00	5.211[

8 rows × 103 columns

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?

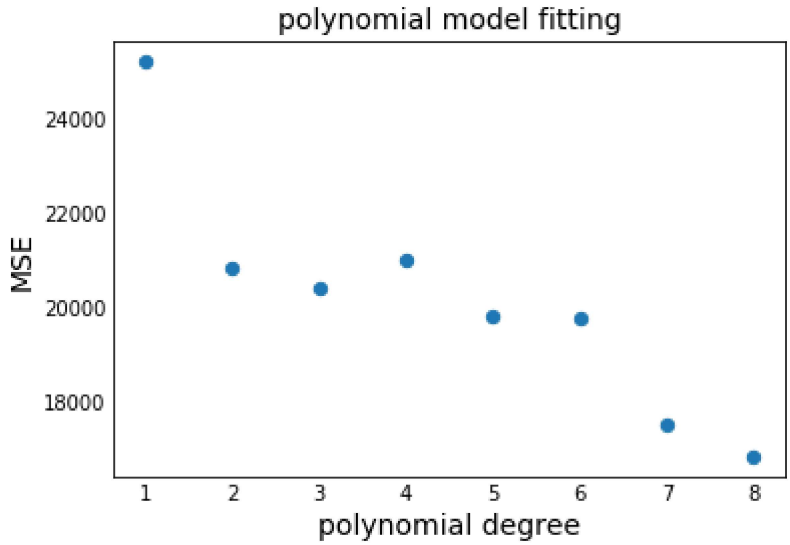
Solutions

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 [24]:
```
# fit OLS model on training data (train) and evaluate model on validation data (v
def fit_poly_models(train, val, max_degree=8):

    # intialize placeholders
    models = []
    MSE = []

    # iterate over all model degrees
    for deg in np.arange(max_degree)+1:

        # build design matrices up to the deg power with interactors
        x_train, y_train, x_val, y_val = get_design_mats(train, val,  deg)

        # train OLS model on training data
        model = OLS(y_train.values, sm.add_constant(x_train.values))
        model = model.fit()

        # generate predictions from validation data and score model with MSE
        pred = model.predict(sm.add_constant(x_val.values))
        MSE.append(mean_squared_error(y_val.values,pred))
        models.append(model)

    return(models, MSE)
```

In [25]:
```
# import data and pre-process
bikes_train, bikes_val = initialize_and_split_bikes("data/bikes_student.csv")

# fit models and calculate MSE
poly_models, MSE = fit_poly_models(bikes_train, bikes_val)

# plot r2 vals
plt.scatter(np.arange(1,9,1),MSE,s=40)
plt.xlabel("polynomial degree",fontsize=14)
plt.ylabel("MSE",fontsize=14)
plt.tick_params(length=0)
plt.title("polynomial model fitting",fontsize=14);
```



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

In [28]:
```
# get the BIC of each model and report
bics = [pm.bic for pm in poly_models]
[print('degree = %i, BIC = %.2f' % (i,b)) for i, b in enumerate(bics)]

# find index of bics with lowest value
best_idx = np.argsort(bics)
print("\nlowest BIC for polynomial model of %i degree(s)" % (best_idx[0]+1))
```

```
degree = 0, BIC = 13361.44
degree = 1, BIC = 13198.15
degree = 2, BIC = 13109.30
degree = 3, BIC = 13128.06
degree = 4, BIC = 13121.26
degree = 5, BIC = 13093.08
degree = 6, BIC = 13035.76
degree = 7, BIC = 13039.46

lowest BIC for polynomial model of 7 degree(s)
```

The first and most obvious trend is that MSE generally decreases with polynomial degree. However, MSE also generally tends to decrease with the number of predictors; so it's not good enough to just choose the model with the lowest MSE. One option is to calculate the Bayesian Information

Criterion (BIC) for each model, which positively weights good model performance but penalizes adding additional parameters. Since polynomial degrees have a natural ordering to them, we don't have to test all possible combinations of predictors. We can find the lowest degree model with the best score. Therefore, I'll choose the model with 7th degree polynomials with the minimum BIC.

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 [29]:
```
# fit ridge regression models to training data (train) and evaluate MSE on valida
# and build data frame to store MSE for model fit on each combindation of lambda
# with terms up to the max_degree-th power
def fit_ridge_models(train, val, lambdas=(.01,.05,.1,.5,1,5,10,50,100), max_degre

    # intialize placeholders
    ridge_MSE = dict()

    # iterate over all values of lambda
    for lam in lambdas:

        # create new column name
        s = r'$\lambda$=' + str(lam)
        MSE = []

        # iterate over all model degrees
        for deg in np.arange(max_degree)+1:

            # intialize new design matrix up to the deg power and fit model
            x_train, y_train, x_val, y_val = get_design_mats(train, val,  deg)
            ridge_m = Ridge(alpha = lam)
            ridge_m.fit(x_train.values, y_train.values)

            # generate predictions from validation data, score model with MSE
            pred = ridge_m.predict(x_val.values)
            MSE.append(mean_squared_error(y_val.values,pred))

        # store results in dict entry
        ridge_MSE[s] = MSE

    ridge_table = pd.DataFrame.from_dict(ridge_MSE)
    ridge_table.rename(dict(zip(np.arange(max_degree), ["degree=" +str(i) for i i


    return(ridge_table)
```

In [30]:
```
bikes_train, bikes_val = initialize_and_split_bikes("data/bikes_student.csv")
with warnings.catch_warnings():
    warnings.simplefilter("ignore")
    MSE=fit_ridge_models(bikes_train, bikes_val);
```

In [31]:
```
# reporting mean squared error in tabular format
pd.options.display.float_format = '{:,.2f}'.format
MSE
```

Out[31]:

	λ=0.01	λ=0.05	λ=0.1	λ=0.5	λ=1	λ=5	λ=10	λ=50	
degree=1	25,227.07	25,142.91	25,069.12	24,829.35	24,730.28	24,603.50	24,621.87	24,945.04	25,3
degree=2	20,791.85	20,730.43	20,677.16	20,500.28	20,425.19	20,359.33	20,413.80	20,784.57	21,(
degree=3	20,376.16	20,282.79	20,197.88	19,931.90	19,824.96	19,650.49	19,636.69	19,942.88	20,2
degree=4	20,994.56	20,907.16	20,827.93	20,569.70	20,459.28	20,242.93	20,198.28	20,410.62	20,6
degree=5	19,792.30	19,721.25	19,656.82	19,451.89	19,366.64	19,201.29	19,171.18	19,413.27	19,7
degree=6	19,737.29	19,683.88	19,632.43	19,415.00	19,282.58	18,944.98	18,851.48	19,126.34	19,5
degree=7	17,486.58	17,440.91	17,401.10	17,239.25	17,136.76	16,821.11	16,674.46	16,531.61	16,6
degree=8	16,774.80	16,727.35	16,688.30	16,564.90	16,519.26	16,445.72	16,412.32	16,494.83	16,7

2.4 Find the best-scoring degree and regularization combination.

In [32]:
```
MSE.min().idxmin(axis=1)
```

Out[32]:
```
'$\\lambda$=10'
```

In [34]:
```
# suppress ill-conditioned matrix warning
with warnings.catch_warnings():
    warnings.simplefilter("ignore")

    # intialize new design matrix up to the deg power and fit model
    x_train, y_train, x_val, y_val = get_design_mats(bikes_train, bikes_val, 8)
    ridge_m = Ridge(alpha = 10)
    ridge_m.fit(x_train.values, y_train.values)

    # generate predictions from validation data, score model with MSE
    pred = ridge_m.predict(x_val.values)
    print('r_squared = %0.3f' % r2_score(y_val.values,pred))
```

```
r_squared = 0.567
```

The minimum value is $R^2 \approx 0.567$ when $\lambda = 10$ and the $polynomial\_degree = 8$

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 [35]:
```
# import data and pre-process train_data
bikes_train, bikes_val = initialize_and_split_bikes("data/bikes_student.csv")

# import test data and get standardized design matrices with poly/interactor terms
test_df = pd.read_csv("data/bikes_test.csv", index_col="Unnamed: 0")
x_train, y_train, x_test, y_test = get_design_mats(bikes_train, test_df,  8)
```

In [36]:
```
# suppress ill-conditioned matrix warning
with warnings.catch_warnings():
    warnings.simplefilter("ignore")

    # fit data
    ridge_m = Ridge(alpha = 10)
    ridge_m.fit(x_train.values, y_train.values)

    # generate predictions from validation data
    pred = ridge_m.predict(x_test.values)
    print(r"r_squared = %.3f" % r2_score(y_test.values,pred))
```

```
r_squared = 0.586
```

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?

The test score is actually very close to the validation score. This is not particularly surprising since we only did a single validation for each model and because the validation data was not taken as a subset from the training data. If it had been taken as a subset of the training data, we might expect that the score could be artificially inflated due to overfitting (ie. we'd be training and testing on a subset of the same data). But with the way we did validation here, simply testing a single set of data witheld from the total is procedurally the same as testing on the test set, with the exception that the test data set is larger.

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` ?

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 [103]:
```
# import dfs and create design matrices
train_df = pd.read_csv("data/bikes_student.csv", index_col="Unnamed: 0")
test_df = pd.read_csv("data/bikes_test.csv", index_col="Unnamed: 0")
x_train, y_train, x_test, y_test = get_design_mats(train_df, test_df,  1, add_int
```

In [105]:
```
# fit OLS model
ols_model = OLS(y_train.values, x_train.values).fit()
```

In [106]:
```
# suppress ill-conditioned matrix warning
with warnings.catch_warnings():
    warnings.simplefilter("ignore")

    # initialize object split data into 10 random chunks
    splitter = KFold(10, shuffle=True)

    # do ridge regression with cross-validation at each alpha
    ridge_cv = RidgeCV(alphas=(.1,.5,1,5,10,50,100), cv=splitter)
    ridge_cv.fit(x_train, y_train)
    print("Ridge best alpha = {} ".format(ridge_cv.alpha_))
```

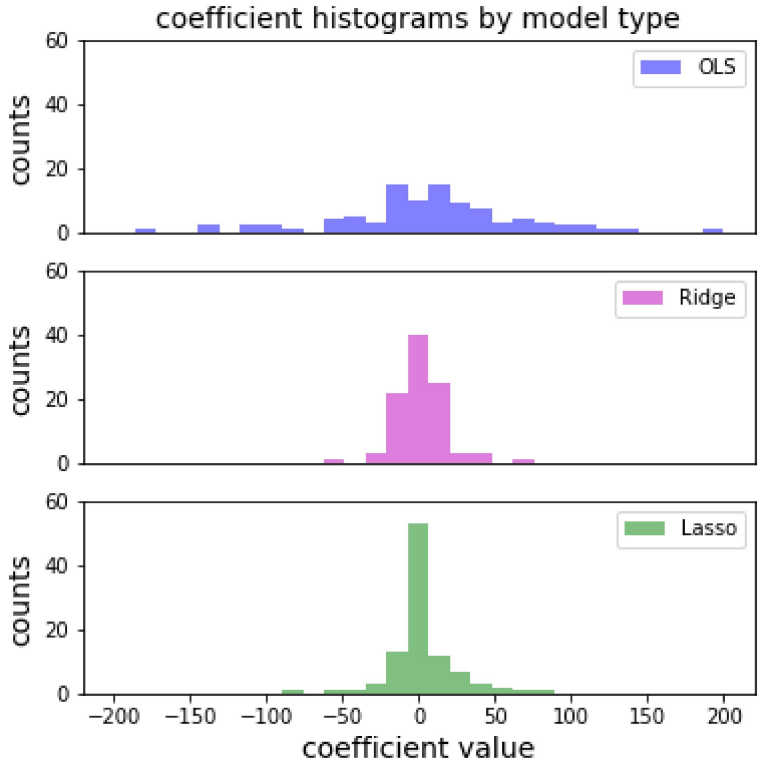
```
Ridge best alpha = 50
```

In [107]:
```
# suppress ill-conditioned matrix warning
with warnings.catch_warnings():
    warnings.simplefilter("ignore")

    # initialize object split data into 10 random chunks
    splitter = KFold(10, shuffle=True)

    # do lasso regression with cross-validation at each alpha (set high max itera
    lasso_cv = LassoCV(alphas=(.1,.5,1,5,10,50,100), cv=splitter, max_iter=100000
    lasso_cv.fit(x_train, y_train)
    print("Lasso: best alpha = {} ".format(lasso_cv.alpha_))
```

```
Lasso: best alpha = 0.5
```

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 [131]:
```
# plotting histograms separately since more than two on the
# same axis looks crowded (and you didn't ask for violin or box plots)
height = 60
plt.figure(figsize=(6,6))
bins = np.linspace(-200,200,30)
plt.subplot(3,1,1)
plt.title('coefficient histograms by model type', fontsize=14)
plt.hist(ols_model.params, bins=bins, color="b", alpha=0.5, label="OLS")
plt.ylim(0,height)
plt.ylabel("counts", fontsize=14)
plt.xticks(())
plt.legend()
plt.subplot(3,1,2)
plt.hist(ridge_cv.coef_, bins=bins, color="m", alpha=0.5, label="Ridge")
plt.ylim(0,height)
plt.ylabel("counts", fontsize=14)
plt.xticks(())
plt.legend()
plt.subplot(3,1,3)
plt.hist(lasso_cv.coef_, bins=bins, color="g", alpha=0.5, label="Lasso")
plt.ylim(0,height)
plt.xlabel("coefficient value", fontsize=14)
plt.legend()
plt.ylabel("counts", fontsize=14);
```



OLS has more predictors with extreme values of the coefficients. This is what we expect since the coefficients magnitudes are unconstrained. Both Ridge and Lasso regression have a pileup of coefficients close to zero since the parameters are constrained by λ. The peak at zero is slightly

larger for lasso regression as we might expect since lasso uses an L1 norm for the cost function and is therefore more likely to force less important parameters towards zero. Also as we might expect, even though lasso has a larger peak at zero, it has slightly longer tails than the ridge histogram since it is more likely to permit some extreme (although rare) coefficient values for the stronger predictors. One plausible reason for this is multicolinearity in the data. As lasso removes colinear predictor(s) from the model, it allows the remaining correlated feature(s) to capture some of the effect it was previously sharing.

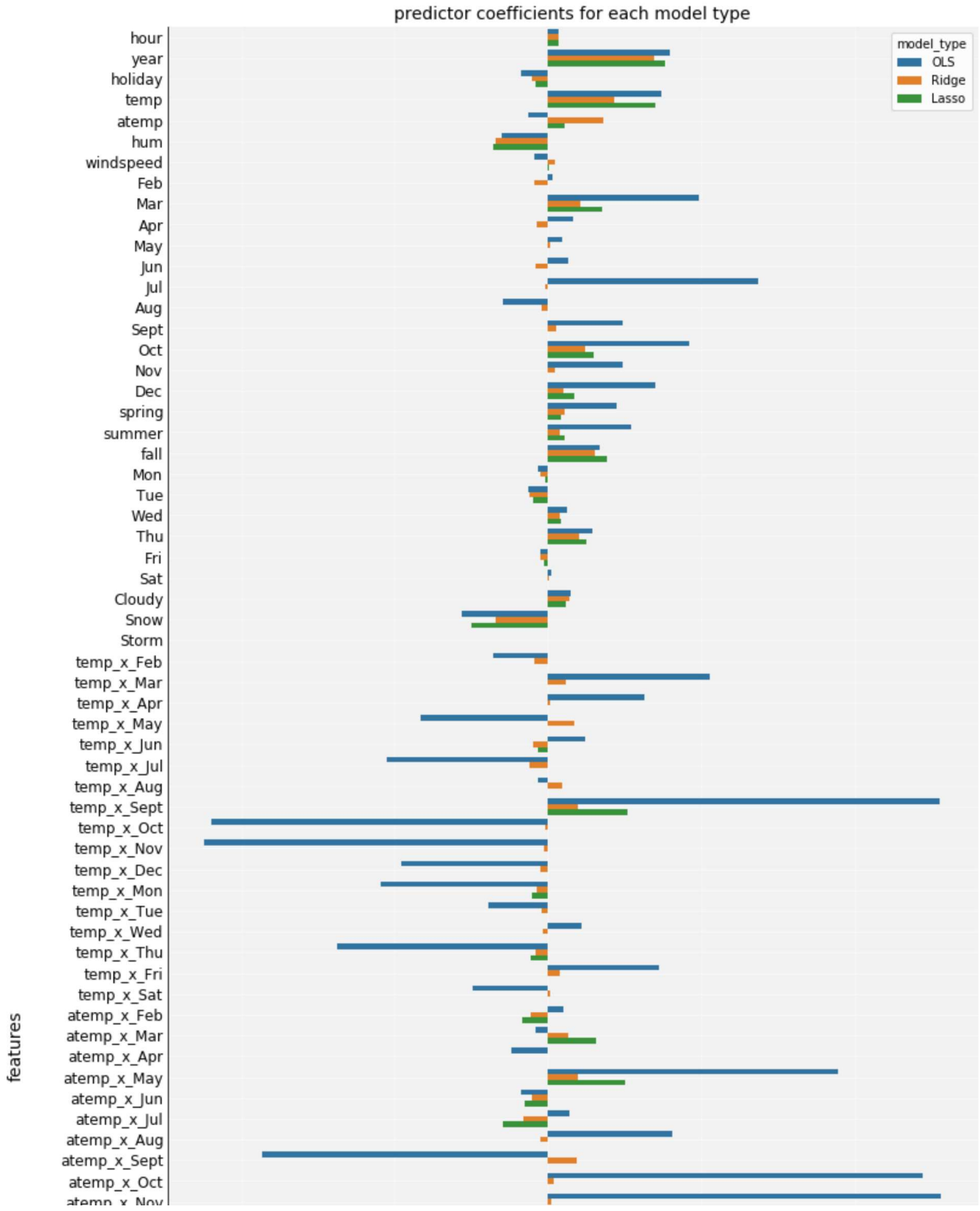
3.3 The plots above show the overall distribution of coefficient values in each model, but do not show how each model treats individual coefficients. Build a plot which cleanly presents, for each feature in the data, 1) The coefficient assigned by OLS, 2) the coefficient assigned by ridge, and 3) the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

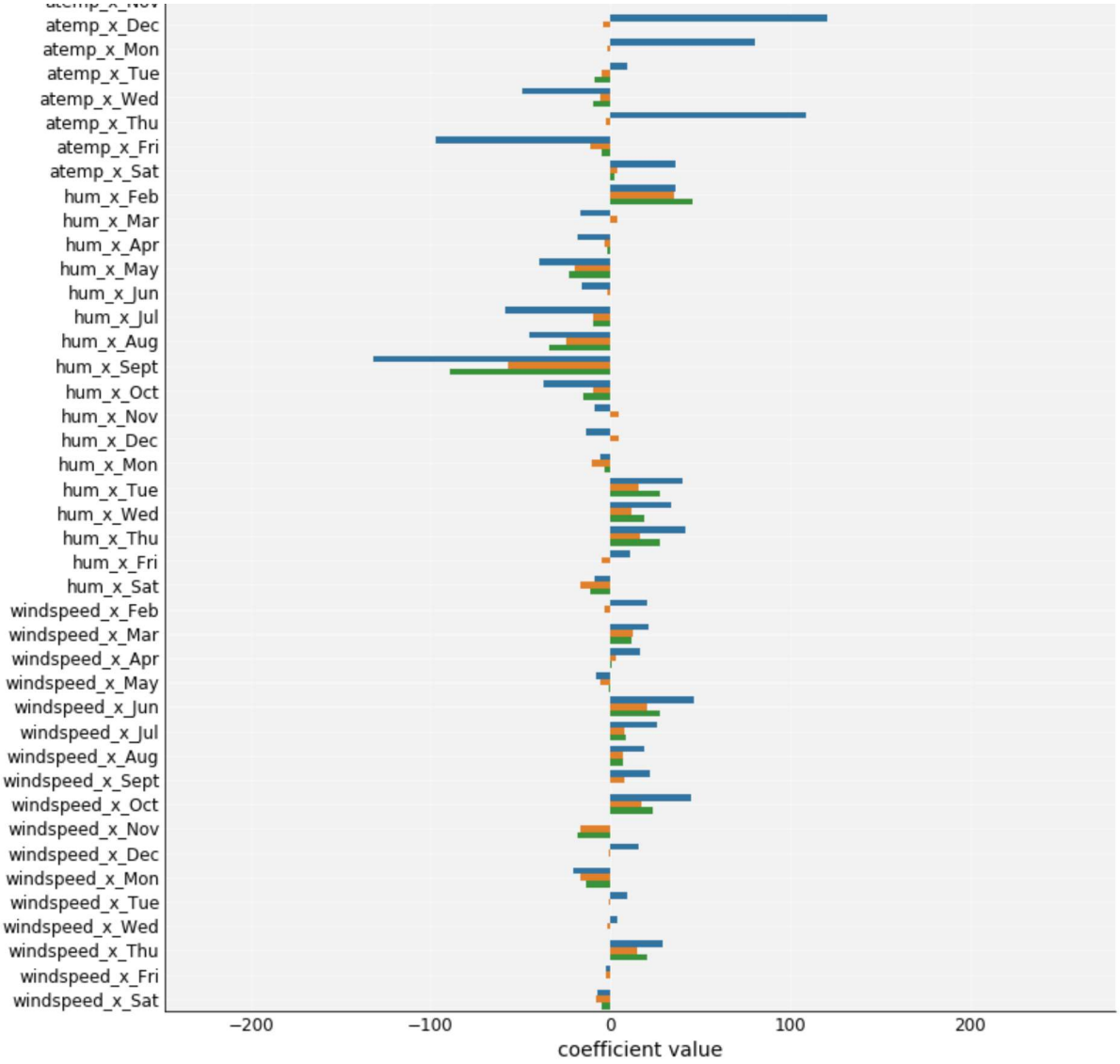
Hint: Bar plots are a possible choice, but you are not required to use them

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

In [110]:
```
# create dataframe for visualization
data = {
    "name": np.tile(x_train.columns.values,3),
    "coef": np.concatenate((ols_model.params, ridge_cv.coef_, lasso_cv.coef_)),
    "model_type": np.array((["OLS"]*x_train.shape[1], ["Ridge"]*x_train.shape[1],
}

features_df = pd.DataFrame.from_dict(data)
```

In [134]:
```
# create new barplot with coefficients color-coded by the type of model
# make it horizontal so feature labels can be read easily
plt.figure(figsize=(12,31))
plt.title("predictor coefficients for each model type", fontsize=14)
ax = sns.barplot(x="coef", y="name", data=features_df, hue="model_type", orient="
sns.despine()
plt.grid(which='both', color='w', linestyle='-', linewidth=0.5, axis='both')
ax.set_facecolor((.95,.95,.95))
ax.set_axisbelow(True)
plt.tick_params(length=0, labelsize=12)
plt.ylabel("features", fontsize=14)
plt.xlabel("coefficient value", fontsize=14);
```





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

The first and most obvious trend is that OLS produces more high-magnitude coefficients than either lasso or ridge regression. The second trend is that where OLS and ridge produce fit small coefficients, lasso shrinks many of the coefficients toward zero. Another trend is the general attenuation of ridge coefficients. The ridge coeficients are nearly always smaller than the OLS coefficients and are also generally smaller than the lasso coefficients when the lasso coefficients are non-zero.

In OLS and Lasso, the `temp` coefficients capture some of the effect of `atemp` while Ridge seems to split the effect evenly between the two features. My prediction is that as lambda increases in value, lasso would push one of these features to zero and put all the weight of the two features into the remaining one. As lambda increases in ridge regression, I would expect that the coefficients of both features would shrink together.

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 a function is definitely useful for the analysis because 1.) it allows the same code to be re-deployed later with a single line of code and 2.) it allowed me to build in customizable functionality that makes it possible to use the code in slightly different ways (eg. variable polynomial features, enabling/disabling interaction features).

One obvious issue that you could encounter just copying and pasting the code is that the functionality can't be checked/edited in one place. You might realize later you made a mistake originally writing the code or that you want to try out a slightly different approach, and then you would have to go back and change it everywhere. Not only is that tedious, but it makes it much easier to make a mistake. In some cases, I wrote convenience functions simply to ensure that the code was deployed consistently.

Writing a function to construct a design matrix from a dataframe is not a bad idea, but a good function would take a more general and flexible form than the one written here (eg. the polynomial column names are hard-coded). One thing that became clear very quickly is that there are a lot of different ways to build a design matrix. For that reason, another sensible approach would be to write a design_matrix class that constructs a design matrix from a dataframe with associated methods for building new features, pre-processing the data etc.

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?

The benefit of regularization is that it allows us to tune the relative importance of variance over bias in our parameter estimation by setting some constraint on the magnitude of the coefficients. This adds stability to our model since it it constrains how tightly the model can be fit to a particular training set. OLS minimizes the bias but tends to produce high variability in parameter estimation, particularly at higher degrees of polynomials, which can lead to overfitting. For example, coefficient magnitudes tend to increase as a function of the degree because higher degrees can generate lines that can increasingly snake through every point in the training data. Regularization is an attempt to place constraints on the model that prevent it from doing this.

Forward selection on features of an OLS model can also be very computationally expensive (order of 2^j for j features), particularly when there is no inherent ordering to the features. Regularization, can be less computationally expensive since it does not require us to fit the model on many different combinations of features.

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?

The appropriateness of the model really depends on the goal. If our goal was purely predictive (eg. develop an algorithm that automatically adjusted fare price on a minute-to-minute basis like that of Uber) it might not matter that it functioned as a black box so long as we could show that it increased revenue.

In the context originally proposed in HW3, the purpose in modeling the data was to gain insights into predicting hourly demand for bike usage that Capital Bikeshare could turn into actionable strategies to increase revenue. That means the model must be interpretable enough for us to make suggestions about how specific features or groups of related features (eg. month) impact the demand for bikes and to communicate those insights to Capital. It would be very problematic if we had an accurate model that was weighted primarily for terms like `hour^4_x_June`. If the model was too complex to be interpreted, I would try to simplify it, either through feature selection or dimensional reduction. We could of course also just not include complicated features that are hard to interpret. All of these strategies may come at the cost of model accuracy.

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

In [109]:
```
# report r_squared on test data for models fit with 1st degree polynomial
ols_y = ols_model.predict(x_test)
ridge_y = ridge_cv.predict(x_test)
lasso_y = lasso_cv.predict(x_test)

print("OLS \t r_squared:\t %.2f" % r2_score(y_test,ols_y))
print("Ridge \t r_squared:\t %.2f" % r2_score(y_test,ridge_y))
print("Lasso \t r_squared:\t %.2f" % r2_score(y_test,lasso_y))
```

```
OLS      r_squared:      0.36
Ridge    r_squared:      0.39
Lasso    r_squared:      0.38
```

I think that the all of the first degree models have predictive value and are reasonably interpretable, though none perform as well as the OLS models fit with higher degree terms. If my goal was to use a model to communicate the general findings, I would consider using any of these 3 models. To that extent, I think the models have potential to offer value to Capital Bikeshare. The sign and magnitude of the coefficients seem to reasonably agree with each other (see plot 3.4 above), and the features used to construct the model are largely things that could be forecast for the week ahead (eg. weather, day, month, season). I would feel confident about using these models to highlight some of the findings of the analysis to offer Capital some intuition for bike demand. If I have to choose a specific model, I would use the lasso regression model since it is less likely to be overfit than OLS and has fewer features and less multicolinearity than the ridge regression model.

However, I don't think these models are ready for production. They perform much more poorly on the test data than the OLS model fit with interactor and polynomial terms in problem 2 ($R^2 \approx 0.6$). Unfortunately, the OLS models fit on 7th and 8th degree terms likely have a lot of multicolinearity. For further tests, I would like to fit the data set with interactor and polynomial terms via Lasso regression due to the high number of predictors in that data set. I also think it would be appropriate to compute the confidence intervals on each of our parameters, search more values of lambda, and increase the number of folds of cross-validation to assess and improve the stability of the model.