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: Haoran Zhao

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-CS
HTML(styles)
```

Out[1]:

import these libraries

In [2]:
```python
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

import seaborn as sns
pd.set_option('display.width', 500)
pd.set_option('display.max_columns', 500)
```

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)](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]:
```
# your code here
bikes_main = pd.read_csv("data/bikes_student.csv", index_col=0).reset_index(d
```

In [4]:
```
bikes_main.head()
```

Out[4]:

	dteday	hour	year	holiday	workingday	temp	atemp	hum	windspeed	casual	registered	c
0	2011-09-07	19	0	0	1	0.64	0.5758	0.89	0.0000	14	212	
1	2012-03-21	1	1	0	1	0.52	0.5000	0.83	0.0896	4	22	
2	2012-08-16	23	1	0	1	0.70	0.6515	0.54	0.1045	58	168	
3	2011-04-28	13	0	0	1	0.62	0.5758	0.83	0.2985	18	103	
4	2012-01-04	0	1	0	1	0.08	0.0606	0.42	0.3284	0	9	

In [5]:
```
bikes_main.describe()
```

Out[5]:

	hour	year	holiday	workingday	temp	atemp	hum
count	1250.000000	1250.000000	1250.000000	1250.000000	1250.000000	1250.000000	1250.00000
mean	11.410400	0.514400	0.030400	0.675200	0.494160	0.473600	0.63844
std	6.885456	0.499993	0.171754	0.468488	0.192529	0.171707	0.18818
min	0.000000	0.000000	0.000000	0.000000	0.040000	0.060600	0.00000
25%	5.000000	0.000000	0.000000	0.000000	0.340000	0.333300	0.50000
50%	11.000000	1.000000	0.000000	1.000000	0.500000	0.484800	0.65000
75%	17.000000	1.000000	0.000000	1.000000	0.660000	0.621200	0.80000
max	23.000000	1.000000	1.000000	1.000000	0.940000	0.909100	1.00000

In [6]:
```
bikes_train, bikes_val = train_test_split(bikes_main, test_size=.2, random_st
```

In [7]:
```
print(bikes_train.shape)
print(bikes_val.shape)
```

```
(1000, 36)
(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 [8]:
```
# your code here
def process(df, columns_to_drop, response):
    y_df = df[response]
    X_df = df.drop(columns_to_drop, axis=1)

    return y_df, X_df
```

In [9]:
```
columns_to_drop = ['counts', 'registered', 'casual', 'workingday', 'month', '
reponse_variable = 'counts'
```

In [10]:
```
practice_y_train, practice_X_train = process(bikes_train, columns_to_drop, re
print(practice_X_train.shape)
print(practice_y_train.shape)
print(practice_X_train.columns)
```

```
(1000, 30)
(1000,)
Index(['hour', 'year', 'holiday', 'temp', 'atemp', 'hum', 'windspeed', 'Feb',
'Mar', 'Apr', 'May', 'Jun', 'Jul', 'Aug', 'Sept', 'Oct', 'Nov', 'Dec', 'sprin
g', 'summer', 'fall', 'Mon', 'Tue', 'Wed', 'Thu', 'Fri', 'Sat', 'Cloudy', 'Sno
w', 'Storm'], dtype='object')
```

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 [11]:
```
# your code here
def standardize(df, target_column, mean=np.inf, std=np.inf):
    if target_column in df.columns:
        df_scaled = df

        if mean == np.inf:
            mean = np.mean(df_scaled[target_column])

        if std == np.inf:
            std = np.std(df_scaled[target_column])

        df_scaled[target_column] = df_scaled[target_column].apply(lambda x: (

        return df_scaled, mean, std

    return
```

In [12]:
```
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']
nonbinary_columns = practice_X_train.columns.difference(binary_columns)
```

In [13]:
```
practice_X_train[nonbinary_columns].describe()
```

Out[13]:

	atemp	hour	hum	temp	windspeed	year
count	1000.000000	1000.000000	1000.000000	1000.000000	1000.000000	1000.000000
mean	0.472546	11.319000	0.639740	0.492780	0.195421	0.509000
std	0.171544	6.879431	0.188386	0.192935	0.125800	0.500169
min	0.060600	0.000000	0.000000	0.040000	0.000000	0.000000
25%	0.333300	5.000000	0.500000	0.340000	0.104500	0.000000
50%	0.484800	11.000000	0.650000	0.500000	0.194000	1.000000
75%	0.621200	17.000000	0.800000	0.660000	0.253700	1.000000
max	0.909100	23.000000	1.000000	0.940000	0.850700	1.000000

In [14]:
```
bikes_main['year'].value_counts()
```

Out[14]:
```
1    643
0    607
Name: year, dtype: int64
```

Since there are only two values for `year` , we can treat this predictor as a binary predictor. Thus, there is no need to standardize `year` column.

In [15]:
```
nonbinary_columns = nonbinary_columns.difference(['year'])
nonbinary_columns
```

Out[15]:
```
Index(['atemp', 'hour', 'hum', 'temp', 'windspeed'], dtype='object')
```

In [16]:
```
practice_X_train_scaled = practice_X_train.copy()

for target_column in nonbinary_columns:
    practice_X_train_scaled, target_column_mean, target_column_stdev = standa
```

In [17]:
```
practice_X_train_scaled[nonbinary_columns].describe()
```

Out[17]:

	atemp	hour	hum	temp	windspeed
count	1.000000e+03	1.000000e+03	1.000000e+03	1.000000e+03	1.000000e+03
mean	4.026779e-15	-1.811190e-16	-4.013789e-15	-4.038325e-15	8.927858e-15
std	1.000500e+00	1.000500e+00	1.000500e+00	1.000500e+00	1.000500e+00
min	-2.402605e+00	-1.646163e+00	-3.397602e+00	-2.347976e+00	-1.554205e+00
25%	-8.121270e-01	-9.189949e-01	-7.421467e-01	-7.922693e-01	-7.231056e-01
50%	7.147176e-02	-4.639332e-02	5.448995e-02	3.744066e-02	-1.130295e-02
75%	8.670022e-01	8.262083e-01	8.511266e-01	8.671507e-01	4.634972e-01
max	2.546131e+00	1.698810e+00	1.913309e+00	2.319143e+00	5.211499e+00

In [18]:
```
practice_X_train.describe()
```

Out[18]:

	hour	year	holiday	temp	atemp	hum	windspeed
count	1000.000000	1000.000000	1000.000000	1000.000000	1000.000000	1000.000000	1000.000000
mean	11.319000	0.509000	0.027000	0.492780	0.472546	0.639740	0.195421
std	6.879431	0.500169	0.162164	0.192935	0.171544	0.188386	0.125800
min	0.000000	0.000000	0.000000	0.040000	0.060600	0.000000	0.000000
25%	5.000000	0.000000	0.000000	0.340000	0.333300	0.500000	0.104500
50%	11.000000	1.000000	0.000000	0.500000	0.484800	0.650000	0.194000
75%	17.000000	1.000000	0.000000	0.660000	0.621200	0.800000	0.253700
max	23.000000	1.000000	1.000000	0.940000	0.909100	1.000000	0.850700

In [19]:
```
practice_X_train_scaled.describe()
```

Out[19]:

	hour	year	holiday	temp	atemp	hum	w
count	1.000000e+03	1000.000000	1000.000000	1.000000e+03	1.000000e+03	1.000000e+03	1.00
mean	-1.811190e-16	0.509000	0.027000	-4.038325e-15	4.026779e-15	-4.013789e-15	8.92
std	1.000500e+00	0.500169	0.162164	1.000500e+00	1.000500e+00	1.000500e+00	1.00
min	-1.646163e+00	0.000000	0.000000	-2.347976e+00	-2.402605e+00	-3.397602e+00	-1.55
25%	-9.189949e-01	0.000000	0.000000	-7.922693e-01	-8.121270e-01	-7.421467e-01	-7.23
50%	-4.639332e-02	1.000000	0.000000	3.744066e-02	7.147176e-02	5.448995e-02	-1.13
75%	8.262083e-01	1.000000	0.000000	8.671507e-01	8.670022e-01	8.511266e-01	4.63
max	1.698810e+00	1.000000	1.000000	2.319143e+00	2.546131e+00	1.913309e+00	5.21

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 [20]:
```
# your code here
def poly(df, target_column, degree):
    tra = PolynomialFeatures(degree, include_bias=False)
    array_poly = tra.fit_transform(df[target_column].values.reshape(-1,1))
    df_poly = pd.DataFrame(array_poly)

    for i in range(degree):
        if i == 0:
            df_poly = df_poly.rename(columns={i: '%s' % target_column})
        else:
            df_poly = df_poly.rename(columns={i: '%s_%i' % (target_column, (i

    return df_poly
```

In [21]:
```
practice_X_train_poly = practice_X_train_scaled.copy().reset_index(drop=True)
ploy_columns = ['temp', 'atemp', 'hum', 'windspeed', 'hour']
practice_X_train_poly = practice_X_train_poly.drop(ploy_columns, axis=1)

for target_column in ploy_columns:
    df_poly = poly(practice_X_train_scaled, target_column, 3)
    practice_X_train_poly = practice_X_train_poly.merge(df_poly, left_index=T
```

In [22]:
```
practice_X_train_poly.head()
```

Out[22]:

	year	holiday	Feb	Mar	Apr	May	Jun	Jul	Aug	Sept	Oct	Nov	Dec	spring	summer	fall
0	1	0	0	0	0	0	0	0	0	0	1	0	0	0	0	1
1	0	0	0	0	0	0	1	0	0	0	0	0	0	0	1	0
2	1	0	0	0	0	0	0	0	1	0	0	0	0	0	1	0
3	1	0	0	0	0	0	0	0	0	0	1	0	0	0	0	1
4	1	0	0	0	0	0	0	0	1	0	0	0	0	0	1	0

In [23]:
```
practice_X_train_poly.columns
```

Out[23]:
```
Index(['year', 'holiday', 'Feb', 'Mar', 'Apr', 'May', 'Jun', 'Jul', 'Aug', 'Sep
t', 'Oct', 'Nov', 'Dec', 'spring', 'summer', 'fall', 'Mon', 'Tue', 'Wed', 'Th
u', 'Fri', 'Sat', 'Cloudy', 'Snow', 'Storm', 'temp', 'temp_2', 'temp_3', 'atem
p', 'atemp_2', 'atemp_3', 'hum', 'hum_2', 'hum_3', 'windspeed', 'windspeed_2',
'windspeed_3', 'hour', 'hour_2', 'hour_3'], dtype='object')
```

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 [24]:
```
# your code here
def interaction(df, a, b):
    df["%s_%s" % (a, b)] = df[a] * df[b]
    return df
```

In [25]:
```
continuous_columns = ['temp', 'atemp', 'hum', 'windspeed']
month_week_columns = ['Feb', 'Mar', 'Apr', 'May', 'Jun', 'Jul', 'Aug', 'Sept'
                      'Mon', 'Tue', 'Wed', 'Thu', 'Fri', 'Sat']

for c in continuous_columns:
    for mw in month_week_columns:
        interaction(practice_X_train_poly, c, mw)

```

In [26]:
```
practice_X_train_poly.shape
```

Out[26]: (1000, 108)

In [27]:
```
practice_X_train_poly.columns
```

Out[27]:
```
Index(['year', 'holiday', 'Feb', 'Mar', 'Apr', 'May', 'Jun', 'Jul', 'Aug', 'Sep
t',
       ...
       'windspeed_Sept', 'windspeed_Oct', 'windspeed_Nov', 'windspeed_Dec', 'wi
ndspeed_Mon', 'windspeed_Tue', 'windspeed_Wed', 'windspeed_Thu', 'windspeed_Fr
i', 'windspeed_Sat'], dtype='object', length=108)
```

In [28]:
```
practice_X_train_poly.head()
```

Out[28]:

	year	holiday	Feb	Mar	Apr	May	Jun	Jul	Aug	Sept	Oct	Nov	Dec	spring	summer	fall
0	1	0	0	0	0	0	0	0	0	0	1	0	0	0	0	1
1	0	0	0	0	0	0	1	0	0	0	0	0	0	0	1	0
2	1	0	0	0	0	0	0	0	1	0	0	0	0	0	1	0
3	1	0	0	0	0	0	0	0	0	0	1	0	0	0	0	1
4	1	0	0	0	0	0	0	0	1	0	0	0	0	0	1	0

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 [29]:

```
def get_design_mats(train_df, val_df, degree,
                    columns_forpoly=['temp', 'atemp', 'hum', 'windspeed', 'hc
                    target_col='counts',
                    bad_columns=['counts', 'registered', 'casual', 'workingda

    # add code here

    # Step 1 - Process Bad Columns
    y_train, X_train_process = process(train_df, bad_columns, target_col)
    y_val, X_val_process = process(val_df, bad_columns, target_col)
    y_train = y_train.reset_index(drop=True)
    y_val = y_val.reset_index(drop=True)

    # Step 2 - Standardize Bad Columns
    binary_columns = ['holiday', 'workingday', 'Feb', 'Mar', 'Apr', 'May', 'J
                      'Sept', 'Oct', 'Nov', 'Dec', 'spring', 'summer', 'fall'
                      'Wed', 'Thu', 'Fri', 'Sat', 'Cloudy', 'Snow', 'Storm']

    X_train_nonbinary_columns = X_train_process.columns.difference(binary_col
    X_val_nonbinary_columns = X_val_process.columns.difference(binary_columns

    if 'year' in X_train_nonbinary_columns:
        # Since there are only two values for column 'year', we can treat it
        X_train_nonbinary_columns = X_train_nonbinary_columns.difference(['ye

    if 'year' in X_val_nonbinary_columns:
        # Since there are only two values for column 'year', we can treat it
        X_val_nonbinary_columns = X_val_nonbinary_columns.difference(['year']

    X_train_scaled = X_train_process.copy()
    X_val_scaled = X_val_process.copy()
    for target_column in X_train_nonbinary_columns:
        X_train_scaled, X_train_column_mean, X_train_column_stdev = standardi
        X_val_scaled, _, _ = standardize(X_val_scaled, target_column, X_trair

    # Step 3 - Add polynomial terms
    X_train_poly = X_train_scaled.copy().reset_index(drop=True)
    X_train_poly = X_train_poly.drop(columns_forpoly, axis=1)

    X_val_poly = X_val_scaled.copy().reset_index(drop=True)
    X_val_poly = X_val_poly.drop(columns_forpoly, axis=1)

    for target_column in columns_forpoly:
        X_train_column_poly = poly(X_train_scaled, target_column, degree)
        X_train_poly = X_train_poly.merge(X_train_column_poly, left_index=Tru

        X_val_column_poly = poly(X_val_scaled, target_column, degree)
        X_val_poly = X_val_poly.merge(X_val_column_poly, left_index=True, rig

    # Step 4 - Add interaction terms
    continuous_columns = ['temp', 'atemp', 'hum', 'windspeed']
    month_week_columns = ['Feb', 'Mar', 'Apr', 'May', 'Jun', 'Jul', 'Aug', 'S
                          'Mon', 'Tue', 'Wed', 'Thu', 'Fri', 'Sat']

    x_train = X_train_poly.copy()
    x_val = X_val_poly.copy()
```

```
    for c in continuous_columns:
        for mw in month_week_columns:
            interaction(x_train, c, mw)
            interaction(x_val, c, mw)

    return x_train, y_train, x_val, y_val

```

In [30]:

```
x_train, y_train, x_val, y_val \
= get_design_mats(bikes_train, bikes_val, degree=3,
                  columns_forpoly=['temp', 'atemp', 'hum','windspeed', 'hour'
                  target_col='counts',
                  bad_columns=['counts', 'registered', 'casual', 'workingday'

```

In [31]:

```
display(x_train.describe())
```

	year	holiday	Feb	Mar	Apr	May	Jun
count	1000.000000	1000.000000	1000.000000	1000.000000	1000.000000	1000.000000	1000.00000
mean	0.509000	0.027000	0.078000	0.085000	0.082000	0.086000	0.08300
std	0.500169	0.162164	0.268306	0.279021	0.274502	0.280504	0.27602
min	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.00000
25%	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.00000
50%	1.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.00000
75%	1.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.00000
max	1.000000	1.000000	1.000000	1.000000	1.000000	1.000000	1.00000

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.

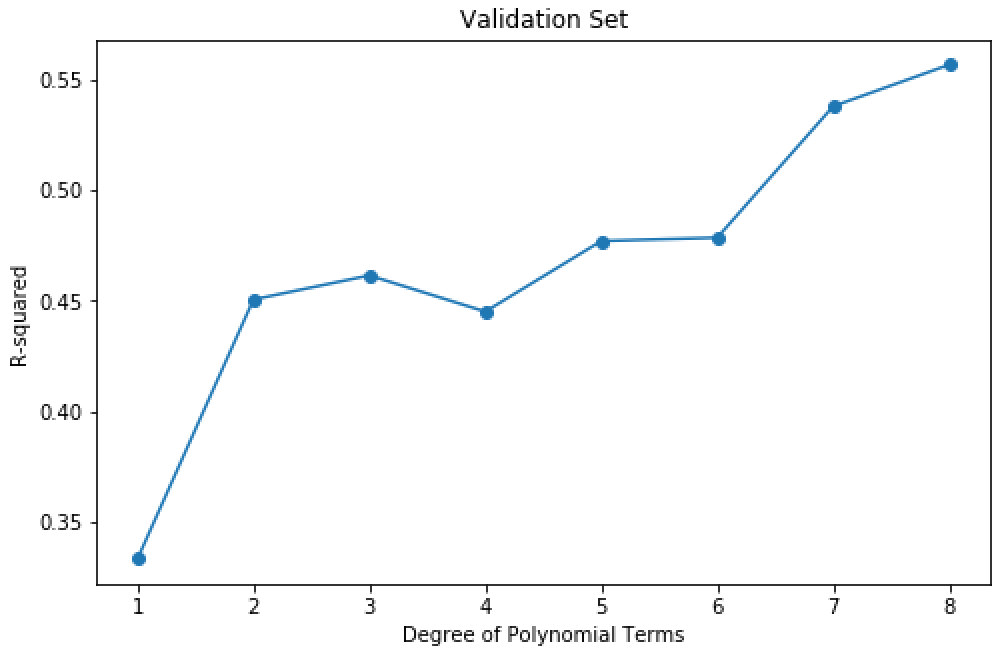
In [32]:
```
# your code here
ols_r2_score = pd.DataFrame(index=range(1, 9), columns=['OLS'])

for d in range(1, 9):
    x_train, y_train, x_val, y_val = \
        get_design_mats(bikes_train, bikes_val, d,
                        columns_forpoly = ['temp', 'atemp', 'hum', 'windspeed
                        bad_columns = ['counts', 'registered', 'casual', 'wor

    X_train = sm.add_constant(x_train)
    OLSModel = sm.OLS(y_train, X_train)
    ols = OLSModel.fit()

    X_test = sm.add_constant(x_val)
    ols_r2_score.loc[d, 'OLS'] = r2_score(y_val, ols.predict(X_test))

```

In [33]:
```
ols_r2_score
```

Out[33]:

	OLS
1	0.333359
2	0.450571
3	0.46147
4	0.445117
5	0.477027
6	0.478536
7	0.537901
8	0.556701

In [34]:
```
plt.figure(figsize=(8, 5))
plt.plot(ols_r2_score.index, ols_r2_score.OLS, '-o')
plt.ylabel('R-squared')
plt.xlabel('Degree of Polynomial Terms')
plt.title('Validation Set')
plt.show()
```



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

your answer here

In general, the higher degree of polynomial models, the higher R-squared in validation set, even though it's not a strict monotonic relationship. I would select the polynomial model with degree of 8, since it has the largest R-squared in validation set.

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 [35]:
```
# your code here
lambdas = [0.01, 0.05, 0.1, 0.5, 1, 5, 10, 50, 100]
ridge_r2_score = pd.DataFrame(index=range(1, 9), columns=lambdas)

for d in range(1, 9):
    for i, lam in enumerate(lambdas):
        x_train, y_train, x_val, y_val = \
            get_design_mats(bikes_train, bikes_val, d,
                            columns_forpoly = ['temp', 'atemp', 'hum', 'winds
                            bad_columns = ['counts', 'registered', 'casual', 

        ridge_reg = Ridge(alpha = lam)
        ridge_reg.fit(x_train, y_train)

        ridge_r2_score.loc[d, lam] = r2_score(y_val, ridge_reg.predict(x_val)
```

In [36]:
```
ridge_r2_score
```

Out[36]:

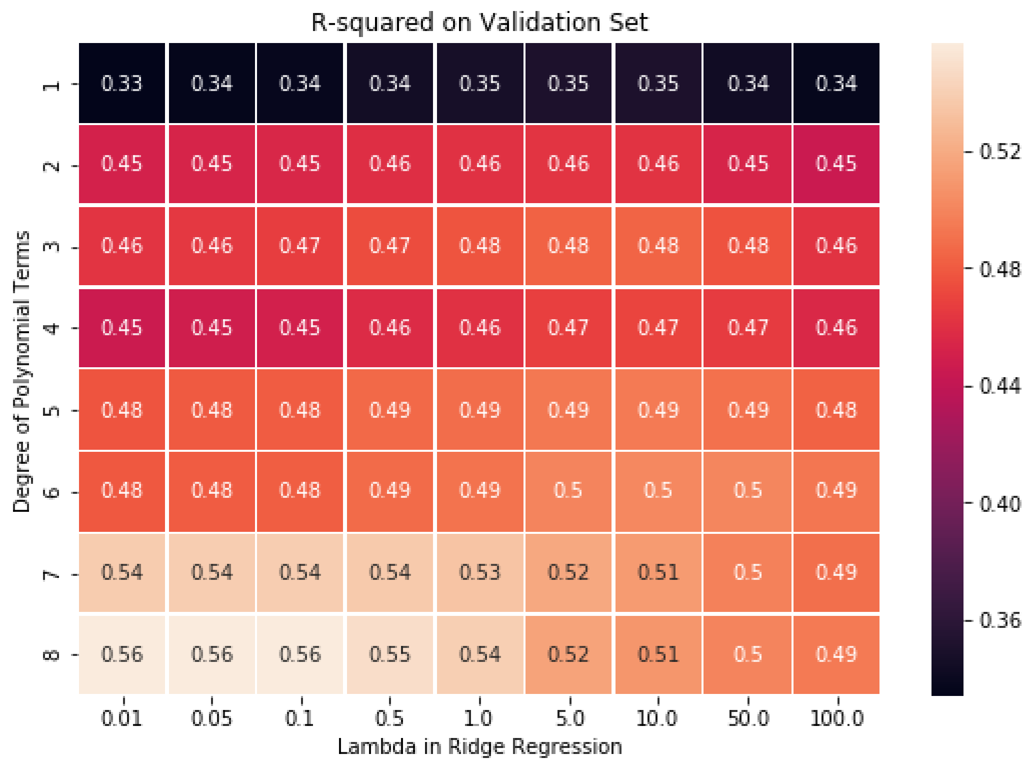
	0.01	0.05	0.1	0.5	1.0	5.0	10.0	50.0	100.0
1	0.33408	0.336304	0.338255	0.344609	0.347257	0.350853	0.350665	0.344274	0.337174
2	0.451156	0.45278	0.454188	0.458877	0.460883	0.462801	0.461559	0.452771	0.445239
3	0.462134	0.464618	0.466883	0.474077	0.477105	0.483068	0.484522	0.476278	0.461331
4	0.445811	0.448144	0.450266	0.457311	0.460495	0.467925	0.470455	0.465925	0.455165
5	0.477542	0.479423	0.48113	0.486581	0.488878	0.493585	0.494745	0.490543	0.483573
6	0.479002	0.480447	0.48185	0.48788	0.491549	0.49981	0.501843	0.500346	0.493281
7	0.538236	0.538785	0.539043	0.537557	0.534172	0.518382	0.511613	0.498093	0.489082
8	0.556881	0.556866	0.556316	0.548542	0.539966	0.515121	0.508435	0.500018	0.494607

In [37]:

```python
ridge_r2_score = ridge_r2_score.astype(float)

f, ax = plt.subplots(figsize=(9, 6))
sns.heatmap(ridge_r2_score, annot=True, linewidths=.5, ax=ax)
ax.set_ylabel('Degree of Polynomial Terms')
ax.set_xlabel('Lambda in Ridge Regression')
ax.set_title('R-squared on Validation Set')
```

Out[37]:

```
Text(0.5,1,'R-squared on Validation Set')
```



2.4 Find the best-scoring degree and regularization combination.

In [38]:

```python
# your code here
ridge_r2_score.columns = ridge_r2_score.columns.map(str)
print("Best ridge model is with lambda of %s and polynomial degree of %s." %
      (ridge_r2_score.max().idxmax(), str(ridge_r2_score[ridge_r2_score.max()
```

```
Best ridge model is with lambda of 0.01 and polynomial degree of 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 [39]:

```
# your code here
bikes_test = pd.read_csv("data/bikes_test.csv", index_col=0).reset_index(drop
bikes_test.head()
```

Out[39]:

	dteday	hour	year	holiday	workingday	temp	atemp	hum	windspeed	casual	registered	co
0	2011-12-03	3	0	0	0	0.24	0.2424	0.70	0.1343	4	5	
1	2011-01-05	22	0	0	1	0.18	0.1970	0.55	0.1343	1	41	
2	2011-02-01	14	0	0	1	0.22	0.2576	0.80	0.0896	5	49	
3	2012-05-29	10	1	0	1	0.74	0.6970	0.70	0.2985	67	116	
4	2011-11-03	22	0	0	1	0.40	0.4091	0.82	0.0000	21	116	

In [41]:

```
# your code here
d = 8
lam = 0.01

x_train, y_train, x_test, y_test = \
    get_design_mats(bikes_main, bikes_test, d,
                    columns_forpoly = ['temp', 'atemp', 'hum', 'windspeed', '
                    bad_columns = ['counts', 'registered', 'casual', 'working

ridge_reg = Ridge(alpha = lam)
ridge_reg.fit(x_train, y_train)

ridge_r2_score_test = r2_score(y_test, ridge_reg.predict(x_test))
print("Best model's R-squared on test data set is %f." % ridge_r2_score_test)
```

```
Best model's R-squared on test data set is 0.586768.
```

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?

your answer here

Actually my model's test score is a little bit better than the validation score. It may happen that the test set is easier to predict, which means the reponse variable is close to our predictions even vs the training set.

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 [42]:
```
# Build the dataset
d = 1
x_train, y_train, x_test, y_test = \
    get_design_mats(bikes_main, bikes_test, d,
                    columns_forpoly = ['temp', 'atemp', 'hum', 'windspeed', '
                    bad_columns = ['counts', 'registered', 'casual', 'working

lambdas = [0.1, 0.5, 1, 5, 10, 50, 100]
```

In [43]:
```
# OLS model
ols = LinearRegression()
ols.fit(x_train, y_train)

ols_r2_score_test = r2_score(y_test, ols.predict(x_test))
print("OLS Model R-squared is %f." % ols_r2_score_test)
```

```
OLS Model R-squared is 0.358744.
```

In [44]:

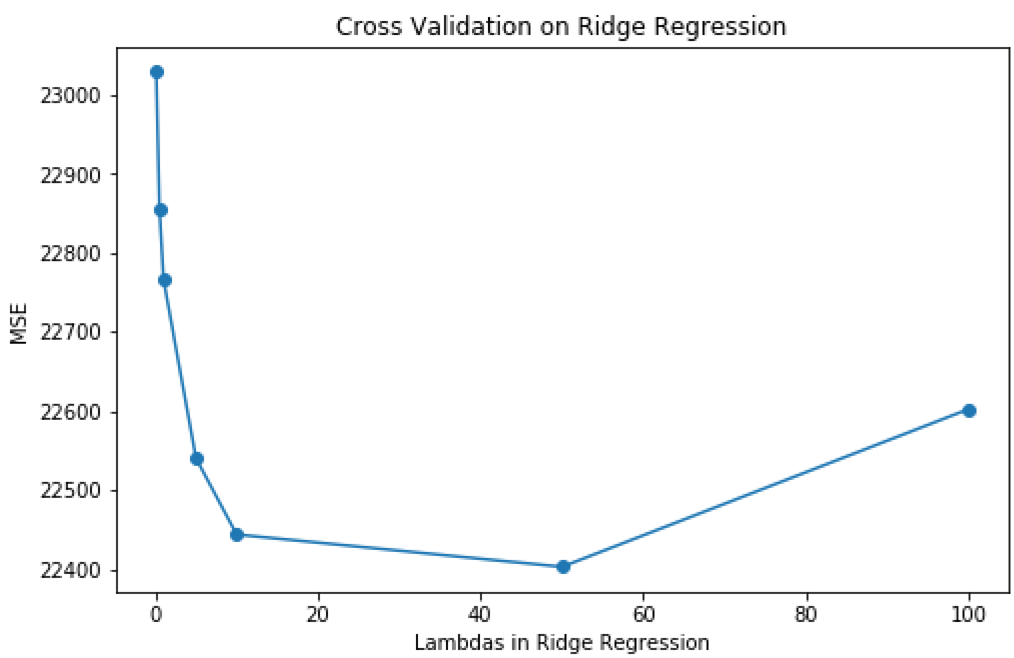
```
# Ridge model with leave one out CV
ridge = RidgeCV(alphas=lambdas, store_cv_values=True).fit(x_train, y_train)

ridge_cv_mse = list(np.mean(ridge.cv_values_, axis=0))
ridge_df = pd.DataFrame({'lambdas': lambdas, 'ridge_cv_mse': ridge_cv_mse})
ridge_df
```

Out[44]:

	lambdas	ridge_cv_mse
0	0.1	23029.365899
1	0.5	22855.986975
2	1.0	22766.062688
3	5.0	22540.141514
4	10.0	22443.606263
5	50.0	22402.675193
6	100.0	22602.414912

In [45]:

```
plt.figure(figsize=(8, 5))
plt.plot(ridge_df.lambdas, ridge_df.ridge_cv_mse, '-o')
plt.ylabel('MSE')
plt.xlabel('Lambdas in Ridge Regression')
plt.title('Cross Validation on Ridge Regression')
plt.show()
```



In [46]:

```
print("The best lambda in Ridge Model is %f." % ridge.alpha_)
```

The best lambda in Ridge Model is 50.000000.

In [47]:

```
ridge_best_lam = 50
ridge_reg = Ridge(alpha = ridge_best_lam)
ridge_reg.fit(x_train, y_train)

ridge_r2_score_test = r2_score(y_test, ridge_reg.predict(x_test))
print("Ridge Model R-squared is %f." % ridge_r2_score_test)
```

```
Ridge Model R-squared is 0.392292.
```

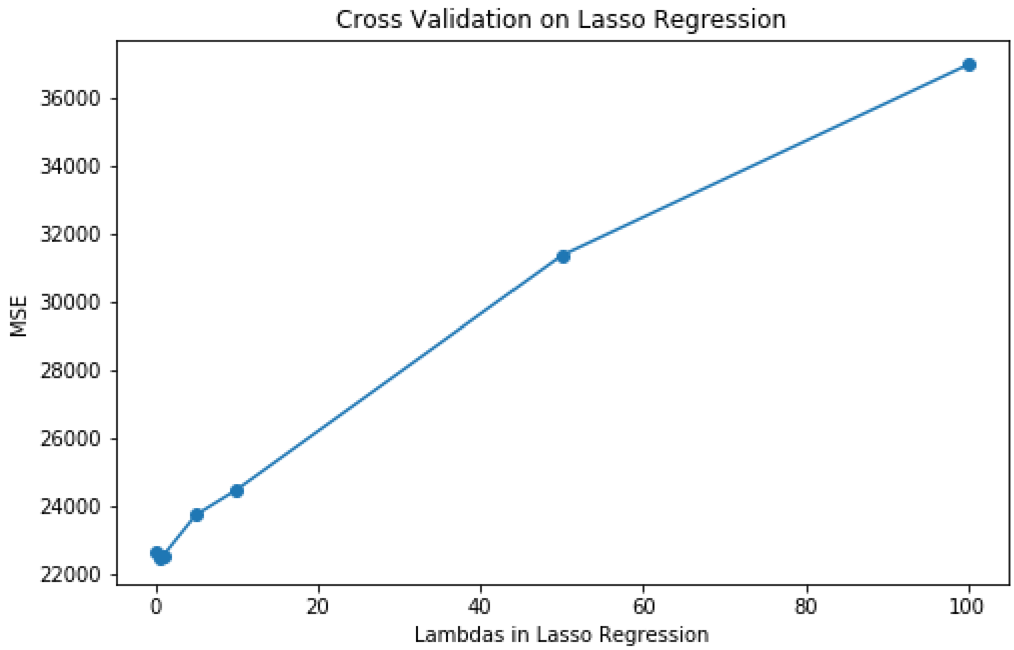
In [48]:

```
# Lasso model with leave one out CV
lasso = LassoCV(alphas=lambdas, cv=x_train.shape[0], max_iter=100000).fit(x_t

df = pd.DataFrame(lasso.mse_path_)
lasso_cv_mse = df.mean(axis=1)
lasso_cv_mse = list(lasso_cv_mse)
lasso_cv_mse.reverse()
lasso_df = pd.DataFrame({'lambdas': lambdas, 'lasso_cv_mse': lasso_cv_mse})
lasso_df
```

Out[48]:

	lambdas	lasso_cv_mse
0	0.1	22635.646820
1	0.5	22422.290237
2	1.0	22526.137886
3	5.0	23721.831984
4	10.0	24460.850929
5	50.0	31364.328704
6	100.0	36990.056302

In [49]:
```python
plt.figure(figsize=(8, 5))
plt.plot(lasso_df.lambdas, lasso_df.lasso_cv_mse, '-o')
plt.ylabel('MSE')
plt.xlabel('Lambdas in Lasso Regression')
plt.title('Cross Validation on Lasso Regression')
plt.show()
```



In [50]:
```python
print("The best lambda in Lasso Model is %f." % lasso.alpha_)
```

```
The best lambda in Lasso Model is 0.500000.
```

In [51]:
```python
lasso_best_lam = 0.5
lasso_reg = Lasso(alpha = lasso_best_lam)
lasso_reg.fit(x_train, y_train)

lasso_r2_score_test = r2_score(y_test, lasso_reg.predict(x_test))
print("Lasso Model R-squared is %f." % lasso_r2_score_test)
```

```
Lasso Model R-squared is 0.381019.
```

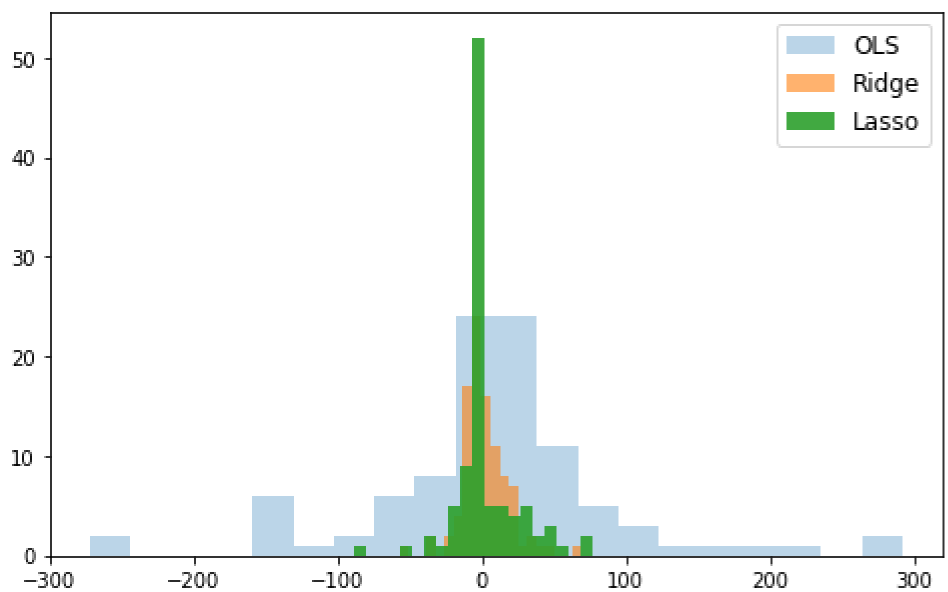
In [52]:
```python
# Model Comparison
print("OLS Model R-squared is %f." % ols_r2_score_test)
print("Lasso Model R-squared is %f." % lasso_r2_score_test)
print("Ridge Model R-squared is %f." % ridge_r2_score_test)
```

```
OLS Model R-squared is 0.358744.
Lasso Model R-squared is 0.381019.
Ridge Model R-squared is 0.392292.
```

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 [53]:
```
# your code here
fig, ax = plt.subplots(1, 1, figsize=(8, 5))
ax.hist(ols.coef_, 20, alpha=0.3, label="OLS")
ax.hist(ridge_reg.coef_, 20, alpha=0.6, label="Ridge")
ax.hist(lasso_reg.coef_, 20, alpha=0.9, label="Lasso")
ax.legend(prop={'size': 12})
```

Out[53]: <matplotlib.legend.Legend at 0x2067184b8d0>



your answer here

Patterns:

- OLS overfits the model that some coefficients of the independent variables are very large, either positive or negative.
- Lasso has a lot of variables with mutted coefficient, which shows as the peak around zero.
- Ridge has a reasonable coefficient distribution without extra value like OLS has, nor many zeros as Lasso has.

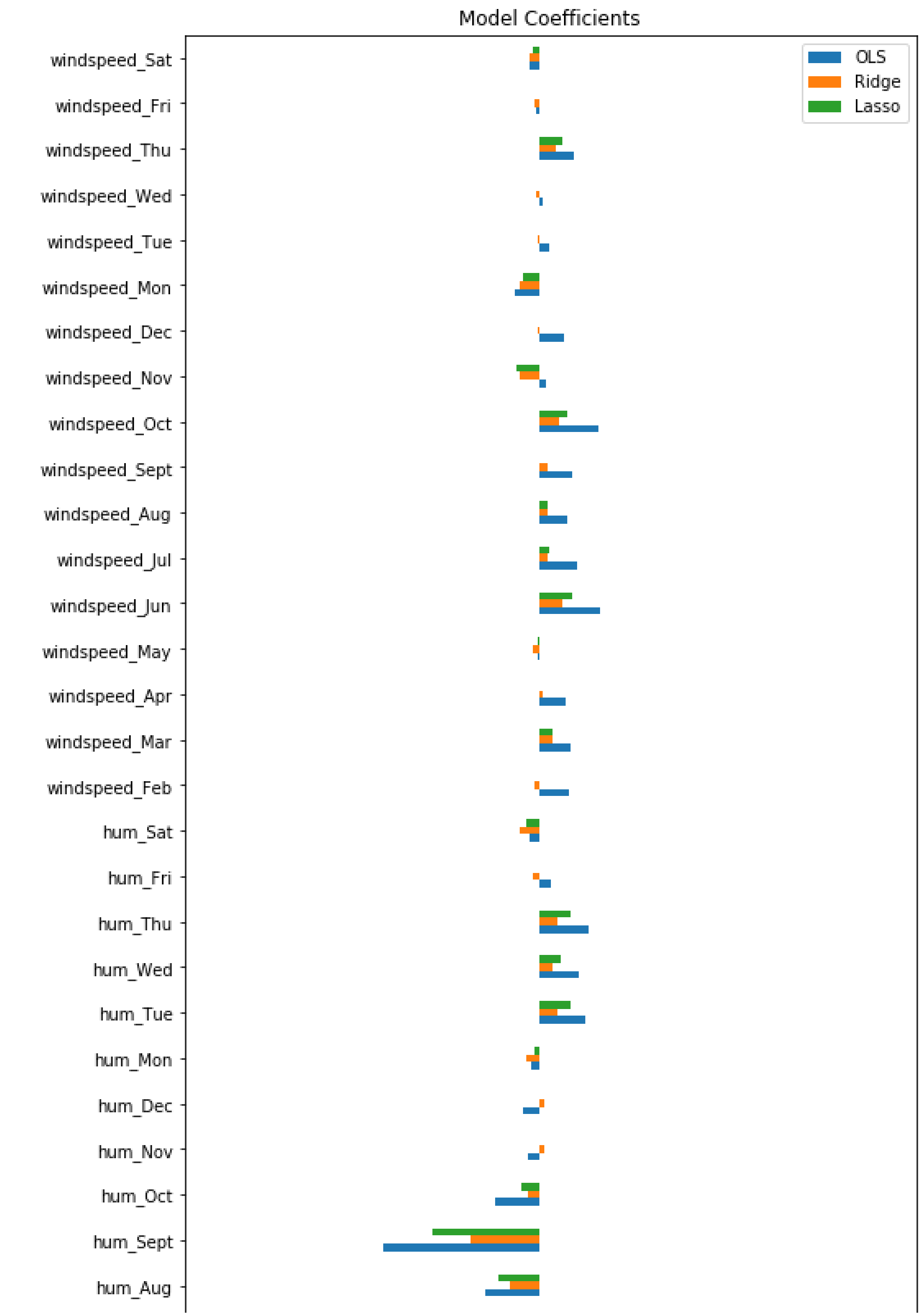
3.3 The plots above show the overall distribution of coefficient values in each model, but do not show how each model treats individual coefficients. Build a plot which cleanly presents, for each feature in the data, 1) The coefficient assigned by OLS, 2) the coefficient assigned by ridge, and 3) the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

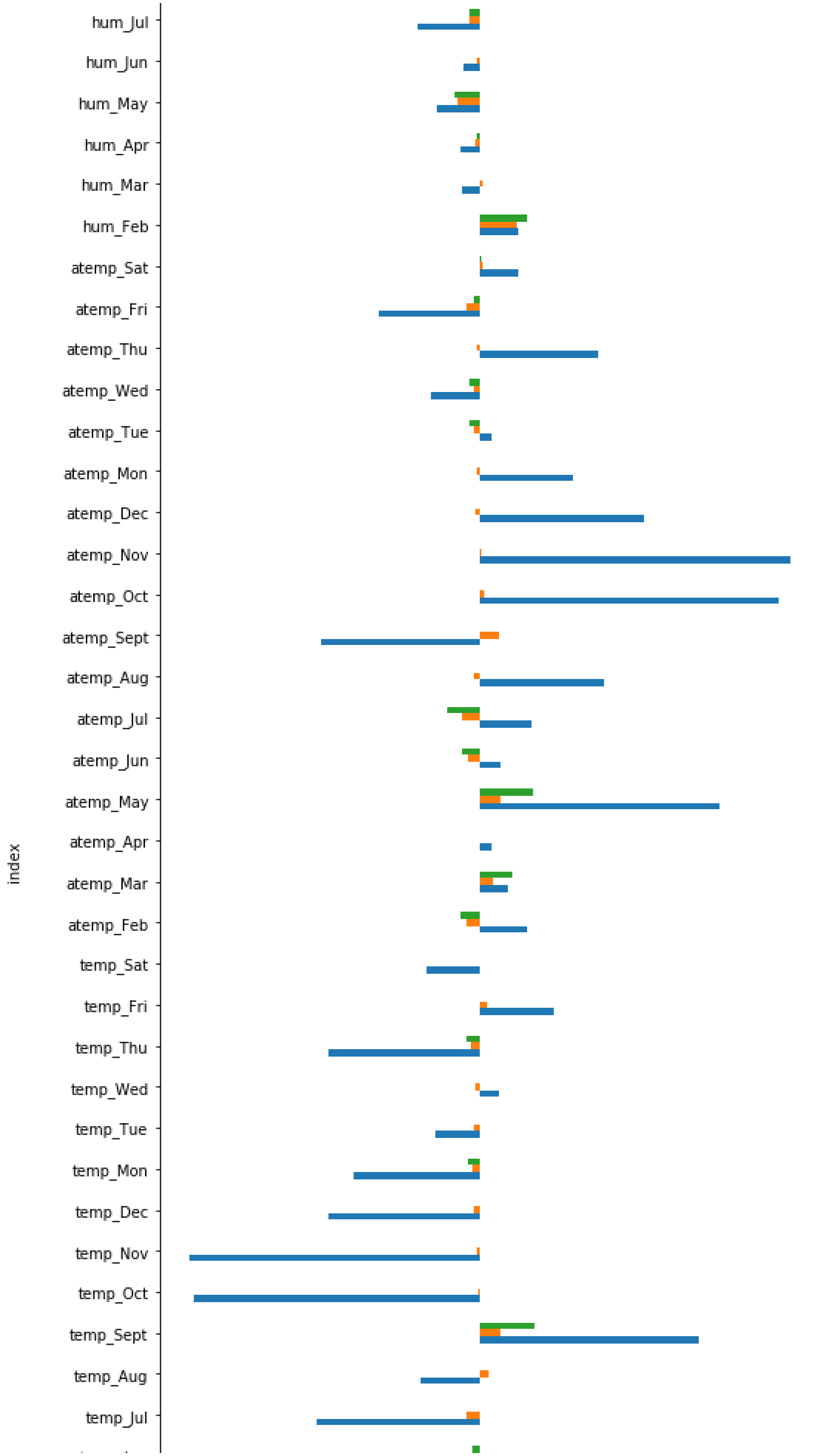
Hint: Bar plots are a possible choice, but you are not required to use them

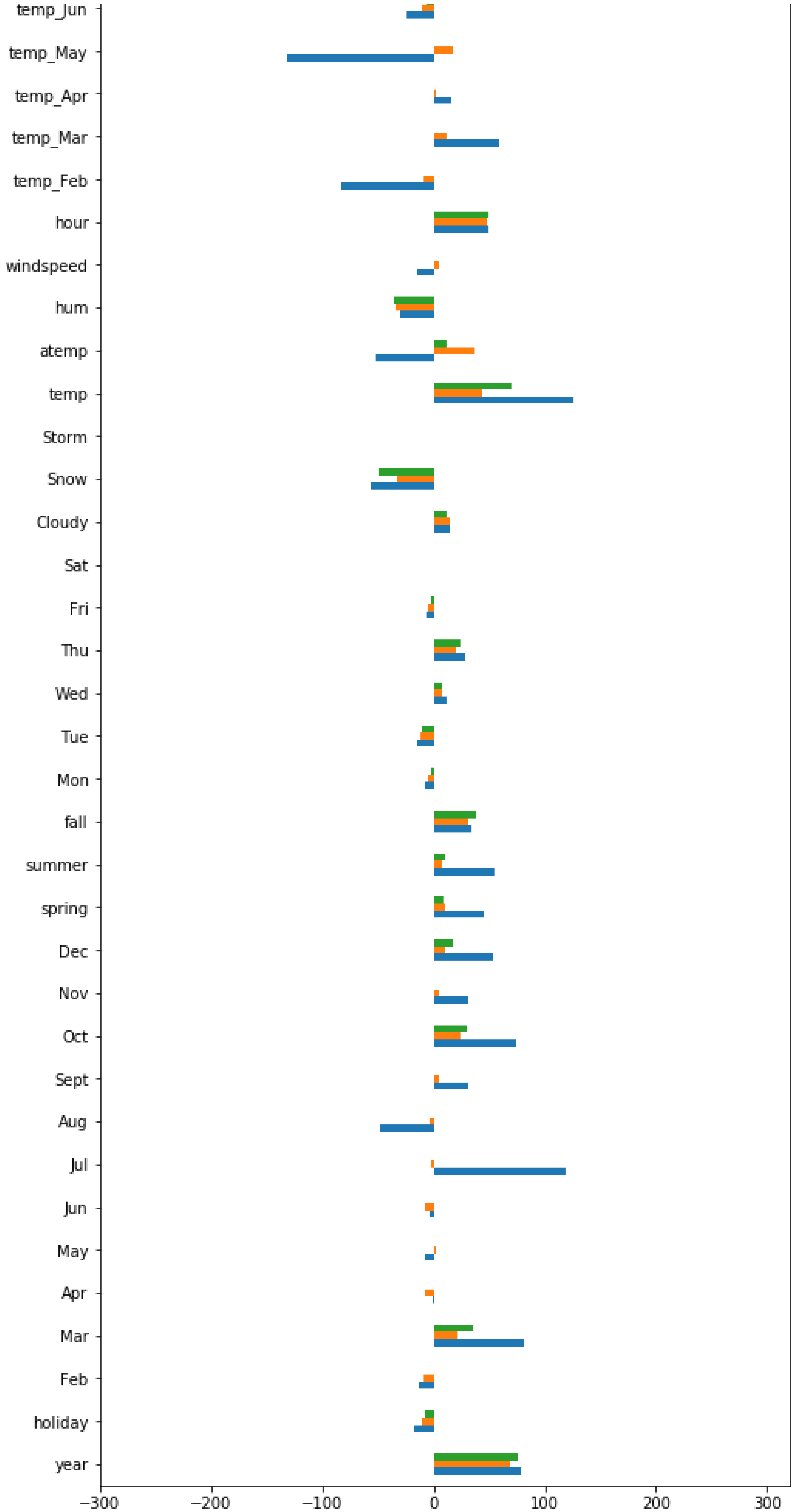
Hint: use `xticks` to label coefficients with their feature names

In [54]:
```
# your code here
df = pd.DataFrame({'index': list(x_train.columns),
                   'OLS': list(ols.coef_),
                   'Ridge': list(ridge_reg.coef_),
                   'Lasso': list(lasso_reg.coef_)})
df = df.set_index('index')
df.plot.barh(figsize=(8, 50), title='Model Coefficients')
```

Out[54]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x2067195c9b0>
```




hum_Jul
hum_Jun
hum_May
hum_Apr
hum_Mar
hum_Feb
atemp_Sat
atemp_Fri
atemp_Thu
atemp_Wed
atemp_Tue
atemp_Mon
atemp_Dec
atemp_Nov
atemp_Oct
atemp_Sept
atemp_Aug
atemp_Jul
atemp_Jun
atemp_May
atemp_Apr
atemp_Mar
atemp_Feb
temp_Sat
temp_Fri
temp_Thu
temp_Wed
temp_Tue
temp_Mon
temp_Dec
temp_Nov
temp_Oct
temp_Sept
temp_Aug
temp_Jul
index


temp_Jun
temp_May
temp_Apr
temp_Mar
temp_Feb
hour
windspeed
hum
atemp
temp
Storm
Snow
Cloudy
Sat
Fri
Thu
Wed
Tue
Mon
fall
summer
spring
Dec
Nov
Oct
Sept
Aug
Jul
Jun
May
Apr
Mar
Feb
holiday
year
−300
−200
−100
0
100
200
300

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

your answer here

Trends:

- OLS overfits the model that some coefficients of the independent variables are very large, either positive or negative.
- Lasso has a lot of variables with zero coefficient.
- Ridge has coefficients for almost all variables but none of them has extra values.
- Most of the variable coefficients have similar signs in three models.

temp and atemp:

- OLS makes atemp coefficient to be very negative, and temp coefficient to be very positive.
- Lasso makes atemp coefficient to be almost zero (slightly positive), and temp coefficient to be very positive.
- Ridge makes both atemp and temp to be both reasonablely positive.

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?

your answer here

Yes, `get_design_mats` function is very useful in the analysis. I don't need to repeatly copy and paste code for processing, standardization, adding polynomial terms and interaction terms for preparing the data set each time before fitting a new model.

Copying and pasting code is bad practice that it makes code (1) development and test very hard, (2) re-usability very low, (3) readability very low, and (4) prone to errors.

`get_design_mat` is wise in general, but it definitely can be improved. For example, `year` has only two values in this problem set and I deal it as a edge case, but the function can take it as an input, so that we don't have to hard code it within the function. Continuous variables and month/week variables can be taken as extra input arguments to the fuction as well.

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?

your answer here

Costs:

- On a relative basis, lasso/ridge is harder to understand vs OLS model, since there is an extra hyperparameter to be setup.
- Computationally, lasso has to be solved with a solver, which takes longer vs a closed form solution of OLS model.

Benefits:

- It's very hard to know which degree of polymomial term is the best in predicting the response variable by specifically setting it up.
- We can setup a large degree of polynomial terms, and let ridge/lasso regularization to tell us which order of which variable is important or not.
- Forward selecting algorithm has a static problem, that the chosen features will never be removed and always kept in the model. Ridge/lasso uses holistic approach to view all the features at the same time.

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?

your answer here

If forecasting ridership as accurately as possible is the ONLY goal, interpretability is not important in this context.

Whether lasso or ridge, as long as it can predicte well, even if the coefficients don't make sense, I would still go with the the model. The obvious reason is that the model is working well and prediction accruracy is the only thing we care about, but it's also possible that the model is capturing some relationship that humans don't easily understand yet.

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

your answer here

I think we have accomplished the goal reasonably well, with R-squared of the three models ranging from 35-39%.

I would recommend Lasso model, because (1) it's clearly better than an overfitted OLS model with higher R-squared on test set, and (2) it has similar R-squared but significantly fewer predictors than Ridge model, so that the BikeShare program management team can take explicit actions to raise revenue, attract more riders, and lower maintenance cost.

Type *Markdown* and LaTeX: α^2

In [ ]:

```
1
```