Artificial Neural Networks

Prof. Dr. Sen Cheng

Oct 28, 2019

Problem Set 4: Model Selection/ Regularization

Tutors: Olya Hakobyan (olya.hakobyan@rub.de), José Donoso (jose.donoso@rub.de)

Further Reading: publications etc, book chapters

Solutions

Core steps needed for all exercises:

```
import numpy as np
import matplotlib.pyplot as plt

# Load the data and separate the predictors and targets
data=np.load('data.npy')
X,Y=data[:,0],data[:,1]

# Generate polynomial features of a given degree
def poly_features(x,degree):
      from sklearn.preprocessing import PolynomialFeatures as poly
      polynomial=poly(degree)
      x_poly=polynomial.fit_transform(x.reshape(len(x),1))
      return  x_poly

# Calculate the mean squared error
def mean_squared_error(y,y_predicted):
    return np.mean((y-y_predicted)**2)

# Plot the fit
def plot_fit_line(ax,x,model,degree):
      x1=np.linspace(min(x),max(x),100)
      x_poly=poly_features(x1,degree)
      ax.plot(x1,model.predict(x_poly))

# Make aspect ratio equal
def set_aspect(ax):

    x0,x1 = ax.get_xlim()
    y0,y1 = ax.get_ylim()
    ax.set_aspect(abs(x1-x0)/abs(y1-y0))

# Initialize the class for linear regression

from sklearn.linear_model import LinearRegression
```

```
model=LinearRegression()
```

1. The holdout method

```
from sklearn.model_selection import train_test_split
x_train,x_test,y_train,y_test=train_test_split(X,Y,test_size=0.2)
degrees=[1,2,3,4,5,6]
train_error=[]
test_error=[]
fig,axes=plt.subplots(2,3)
axes=axes.flatten()

for ind, degree in enumerate(degrees):
    #Get the polynomial features
    train_poly=poly_features(x_train,degree)
    test_poly=poly_features(x_test,degree)

    #Fit the data and assess the error
    model.fit(train_poly,y_train)
    train_error.append(mean_squared_error(y_train,model.predict(train_poly)))
    test_error.append(mean_squared_error(y_test,model.predict(test_poly)))

    #Visualize the data points and fitting lines
    ax=axes[ind]
    plot_fit_line(ax,x_train,model,degree)
    ax.set_title('Degree= %s' %degree,fontsize=14)
    ax.scatter(x_train,y_train,color='r')
    ax.set_xlabel('x',fontsize=14)
    ax.set_ylabel('y',fontsize=14)
    set_aspect(ax)

fig.tight_layout()

# Plot the training and validation errors
fig1,ax1=plt.subplots(1,1)
ax1.plot(degrees,train_error,'bo-',label='Train')
ax1.plot(degrees,test_error,'ro-',label='Test')
ax1.set_xlabel('Polynomial degree',fontsize=14)
ax1.set_ylabel('MSE',fontsize=14)
ax1.legend(prop={'size':13})
```

A simple linear model underfits the data and increasing the polynomial degree reduces the bias.

The holdout method may be a viable option in case of large datasets and limited computational resources. However, as you have seen in this exercise, a big issue with this approach is that your model performance depends on how the data is split between training and validation sets. Especially if your dataset is very small, the exact configuration of the limited training points determines the outcome. In Figure 2 are examples, where the best fit changes with different random states (left to right). In all cases, the bias is reduced with higher polynomial degrees, while the minimum variance, i.e. test error varies across different runs.

(Figure1).

2. Cross-validation

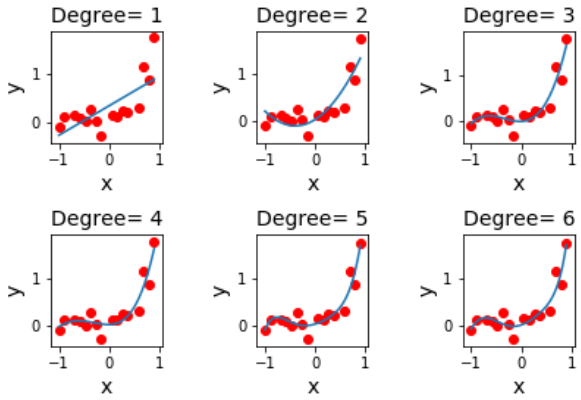


Figure 1: Fits of different polynomials

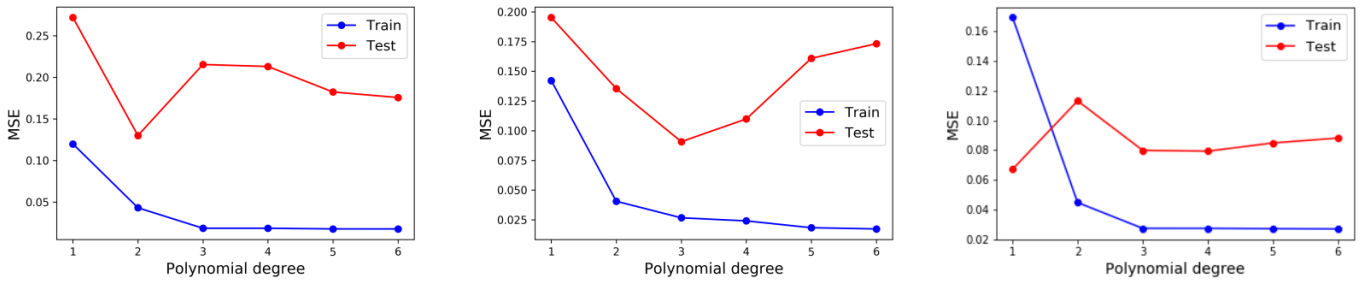
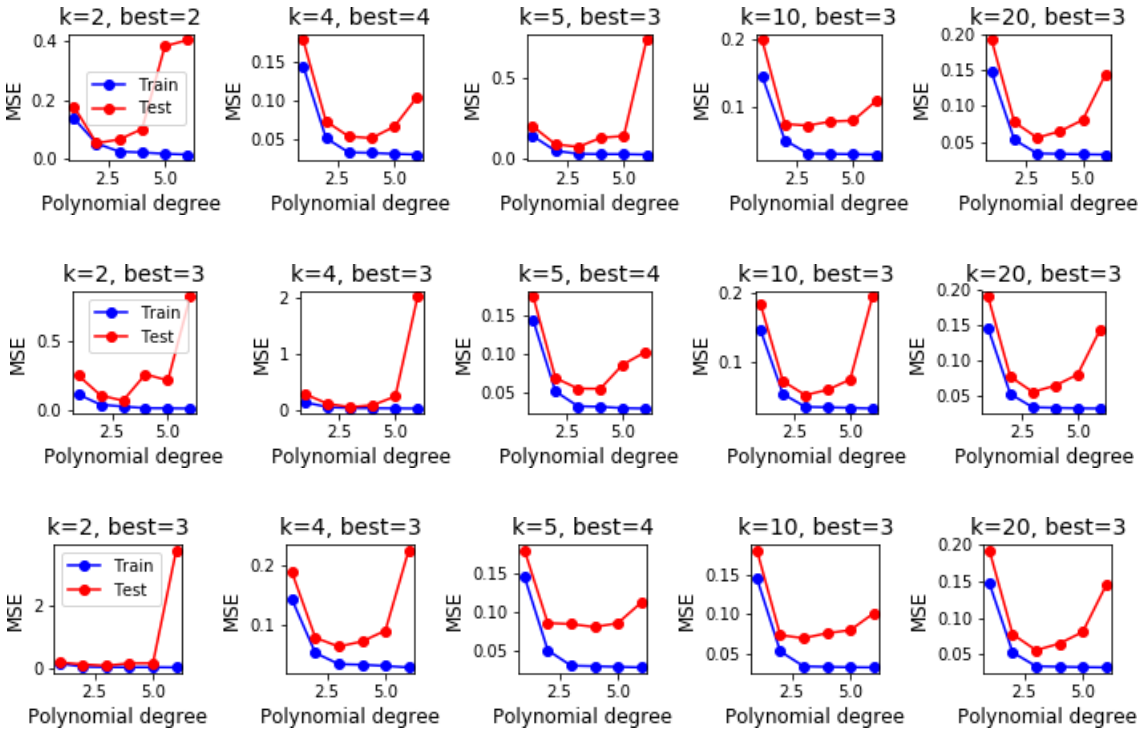


Figure 2: Results from different runs

```
from sklearn.model_selection import KFold
k_folds=[2,4,5,10,20]
degrees=[1,2,3,4,5,6]
fig,axes=plt.subplots(1,len(k_folds),figsize=(10,10))
for ind,k_fold in enumerate(k_folds):
        #get an instance of KFold class
    kf = KFold(n_splits=k_fold,shuffle=True)
    kf.get_n_splits(X)

    #allocated to store the error
    train_fold_err=[[] for j in range(k_fold)]
    test_fold_err=[[] for j in range(k_fold)]
    count=0


    for train_index, test_index in kf.split(X):
        #get the training and test data for consecutive folds
        x_train, x_test = X[train_index], X[test_index]
        y_train, y_test = Y[train_index], Y[test_index]

        #perform regression for different polynomial degrees

        for degree in degrees:
            train_poly=poly_features(x_train,degree)
            test_poly=poly_features(x_test,degree)
            model.fit(train_poly,y_train)
```

```
            train_fold_err[count].append(\
            mean_squared_error(y_train,model.predict(train_poly)))
            test_fold_err[count].append(\
            mean_squared_error(y_test,model.predict(test_poly)))
        count+=1

    #average error across folds and plot
    train_error=np.mean(train_fold_err,axis=0)
    test_error=np.mean(test_fold_err,axis=0)
    ax=axes[ind]
    ax.plot(degrees,train_error,'bo-',label='Train')
    ax.plot(degrees,test_error,'ro-',label='Test')
    ax.set_xlabel('Polynomial degree',fontsize=12)
    ax.set_ylabel('MSE',fontsize=12)
    ax.set_title('Best=%s'%(np.argmin(test_error)+1),fontsize=14)
    set_aspect(ax)
        ax.legend(prop={'size':10})
```

A big advantage of cross-validation method is that no data is wasted, and all data points are used for both training and testing. A rule of thumb for cross-validation is to use 5 or 10 folds. However, you should always keep in mind that the stability of the model decreases if the dataset is small. With really small datasets, it is recommended to use k=n folds, where n refers to the size of the dataset. It is known as leave-one-out cross-validation. This method allows to effectively use the data for training since in each step only one data point is taken for testing. However, for large datasets, this method may be computationally expensive, since a large number of iterations will be needed. Below are examples of the training and test errors vs model complexity for different folds. Each row represents a different run. You can see that most of the time, the 3rd degree polynomial is selected; however, the training and test errors exhibit the most stable pattern when k=n=20.



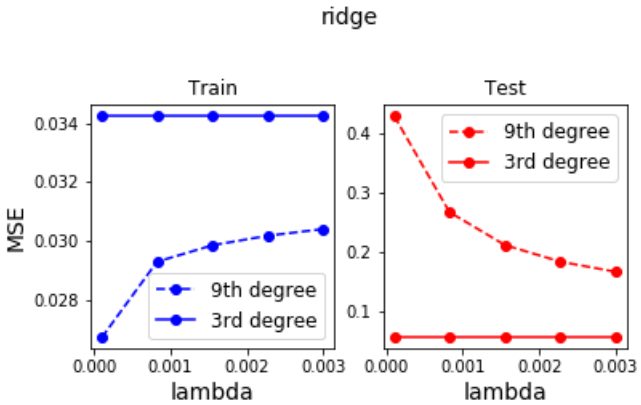
3. Regularization

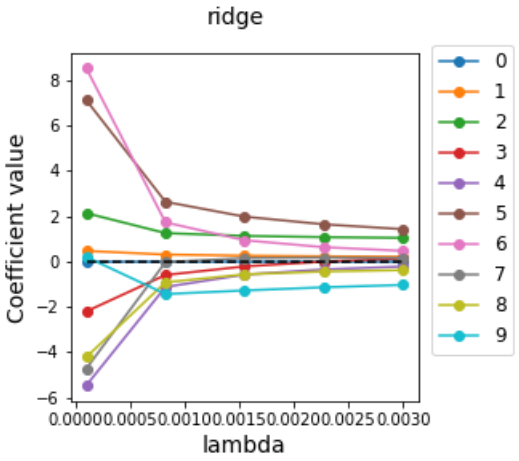
```
# get instances of regularizing models
```

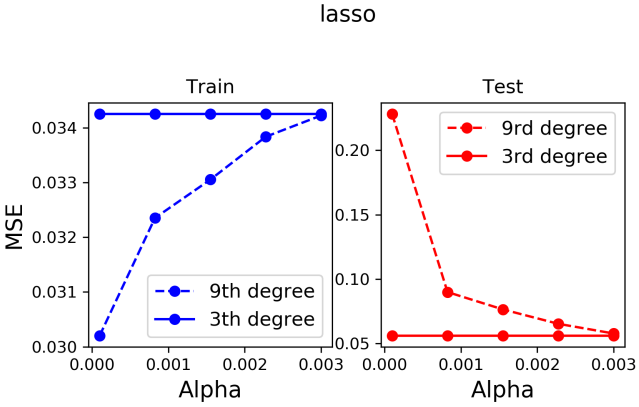
```
regularization='ridge' #'lasso' or 'ridge'
if regularization=='ridge':
    from sklearn.linear_model import Ridge
    reg_model=Ridge()
elif regularization=='lasso':
    from sklearn.linear_model import Lasso
    reg_model=Lasso()


lambdas=np.linspace(0.0001,.003,5)
overfit_degree=9
best_degree=3

# get an instance of KFold class
k=20
kf = KFold(n_splits=k, shuffle=True)

# allocate for storing
train_error=[]
test_error=[]
train_error_reg=[[] for i in range(len(lambdas))]
test_error_reg=[[] for i in range(len(lambdas))]
coefs=[[] for i in range(len(lambdas))]

for train_index, test_index in kf.split(X):
    # get the training and test data for consecutive folds
    x_train, x_test = X[train_index], X[test_index]
    y_train, y_test = Y[train_index], Y[test_index]
    # fit the best-fit polynomial without regularization
    model.fit(poly_features(x_train,best_degree),y_train)
    train_error.append(mean_squared_error(y_train,model.predict \
    (poly_features(x_train,best_degree))))
    test_error.append(mean_squared_error(y_test,model.predict \
    (poly_features(x_test,best_degree))))

    #use regularization with different strengths for overfitting model
    for ind,Lambda in enumerate(lambdas):
        reg_model.alpha=Lambda
        reg_model.fit(poly_features(x_train,overfit_degree),y_train)
        train_error_reg[ind].append(mean_squared_error(y_train, \
                    reg_model.predict(poly_features \
                    (x_train,overfit_degree))))
        test_error_reg[ind].append(mean_squared_error(y_test,\
                    reg_model.predict(poly_features \
                    (x_test,overfit_degree))))
        coefs[ind].append(reg_model.coef_)

# average across folds
train_error=np.mean(train_error)
test_error=np.mean(test_error)
train_error_reg=np.mean(train_error_reg,axis=1)
test_error_reg=np.mean(test_error_reg,axis=1)
coefs=np.mean(coefs,axis=1)
```

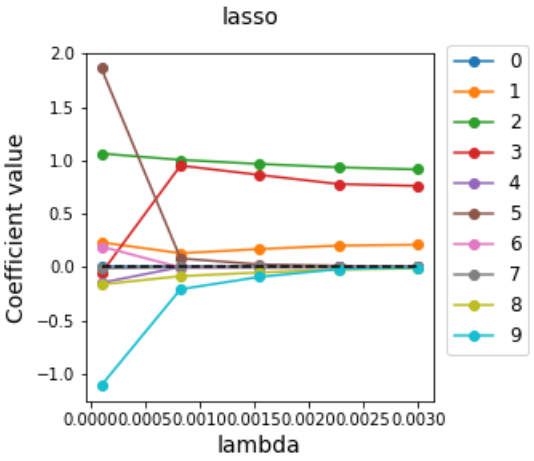
```
#
# plot the errors vs lambda
fig,axes=plt.subplots(1,2)
axes[0].plot(lambdas,train_error_reg,'bo—',label='9th degree')
axes[1].plot(lambdas,test_error_reg,'ro—',label='9th degree')
axes[0].plot(lambdas,[train_error]*len(lambdas),'bo–',label='3rd degree')
axes[1].plot(lambdas,[test_error]*len(lambdas),'ro–',label='3rd degree')
axes[0].set_xlabel('lambda',fontsize=14)
axes[1].set_xlabel('lambda',fontsize=14)
axes[0].set_ylabel('MSE',fontsize=14)
axes[0].set_xlabel('lambda',fontsize=14)
axes[0].legend(prop={'size':12})
axes[1].legend(prop={'size':12})
axes[0].set_title('Train')
axes[1].set_title('Test')
set_aspect(axes[0])
set_aspect(axes[1])
fig.suptitle(regularization,fontsize=14)

# plot coefficients vs lambda
fig1,ax1=plt.subplots(1,1)
for ind,coef in enumerate(coefs.T):
  ax1.plot(lambdas,coef,'o–',label=ind)
ax1.plot(lambdas,[0]*len(lambdas),'k—')
set_aspect(ax1)
ax1.set_xlabel('lambda',fontsize=14)
ax1.set_ylabel('Coefficient value',fontsize=14)
ax1.legend(prop={'size':12},loc='upper right', bbox_to_anchor=(1.3, 1.05))
fig1.suptitle(regularization,fontsize=14)
```

When regularization is applied, the training error of the overfitting model increases with λ, while the test error decreases. Both approach the values of the best-fit. In ridge regression, the coefficients approach zero but don't become zero. Lasso regression performs feature selection by setting irrelevant coefficients to zero, leaving 3 non-zero ones, which had to be expected given that the data is best represented by a 3 degree polynomial. In fact, the function underlying the data is $f(x) = x^3 + x^2 + \varepsilon$,where ε is drawn from a normal distribution with a mean of 0 and standard deviation of 0.4. Since the contribution of these coefficients is eliminated, lasso regression is more successful in approaching the best fit in the current scenario.




ridge
Coefficient value
lambda
0
1
2
3
4
5
6
7
8
9


lasso
Train
Test
MSE
Alpha
Alpha
9th degree
3th degree
9rd degree
3rd degree


lasso
Coefficient value
lambda
0
1
2
3
4
5
6
7
8
9