Lab 5 - ML Programming

December 10, 2021

1 EXERCISE 1

1.1 Backward search for variable selection

[85]:
```
## Import data and necessary libraries
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

bank = pd.read_csv('bank.csv',delimiter=";")
```

[86]:
```
## Convert any non-numeric values to numeric values
## Remove NA/missing values

## Recode month and y columns as numbers to reduce sparsity
bank['month'] = bank['month'].
↪replace(['jan','feb','mar','apr','may','jun','jul','aug','sep','oct','nov','dec'],␣
↪['1','2','3','4','5','6','7','8','9','10','11','12'])
bank['y'] = bank['y'].replace(['no','yes'],['0','1'])

## Separate numeric and categorical columns
numerical_columns = ['age',␣
↪'balance','day','month','duration','campaign','pdays','previous','y']
cat_cols = [c for c in bank.columns if c not in numerical_columns]

## Converting columns that are numerical into numeric dtypes
for col in numerical_columns:
    bank[col]=bank[col].astype(np.float32)

bank_numeric = pd.concat([bank[numerical_columns],pd.
↪get_dummies(bank[cat_cols])],axis=1)
bank_numeric = bank_numeric.dropna()
```

[77]:
```
bank_numeric['y']. value_counts()
```

[77]:
```
0.0    4000
1.0     521
```

```
Name: y, dtype: int64
```

[78]:
```
## Balance the dataset, keeping as much data as possible
n0 = 520
n1 = 520
## Get indexes and lengths for the classes respectively
idx0 = bank_numeric.index.values[bank_numeric['y'] == 0.0]
idx1 = bank_numeric.index.values[bank_numeric['y'] == 1.0]
len0 = len(idx0)   ## 4000
len1 = len(idx1)   ## 521
## Draw randomly from the indices
draw0 = np.random.permutation(len0)[:n0]
idx0 = idx0[draw0]
draw1 = np.random.permutation(len1)[:n1]
idx1 = idx1[draw1]
## Combine the drawn indexes
idx = np.hstack([idx0, idx1])
## Create new dataset
bank_numeric = bank_numeric.loc[idx, :]
## Check length of new dataset (should be 2x520)
bank_numeric['y']. value_counts()
```

[78]:
```
0.0    520
1.0    520
Name: y, dtype: int64
```

[79]:
```
## Split the data into a train/test splits according to the ratios 80%:20%
bank_numeric_train = bank_numeric.sample(frac=0.8,random_state=42) ## random␣
↪state is just a seed value
bank_numeric_test = bank_numeric.drop(bank_numeric_train.index)
```

[80]:
```
## Normalize the data with xi-μ/
## We should first normalize the training data
for column in bank_numeric_train.columns:
    bank_numeric_train[column] =␣
↪(bank_numeric_train[column]-bank_numeric_train[column].mean()
                                 )/bank_numeric_train[column].std()

## To normalize test set, apply normalization parameters obtained from training␣
↪set
for column in bank_numeric_test.columns:
    bank_numeric_test[column] =␣
↪(bank_numeric_test[column]-bank_numeric_train[column].mean()
                                 )/bank_numeric_train[column].std()
```

[ ]:

```python
## Implement logistic regression and mini-batch Gradient Ascent
## Keep learning rate and batch size fixed and iteratively do backward␣
↪selection keeping track of the AIC metric
## Report the final error on test set

## Split data into features and targets
x_train = bank_numeric_train.iloc[:,:-1].values
y_train = bank_numeric_train.iloc[:,-1].values
x_test = bank_numeric_test.iloc[:,:-1].values
y_test = bank_numeric_test.iloc[:,-1].values

def sigmoid(x,beta):
    return 1.0/(1.0 + np.exp(-1*(np.dot(x,beta))))

def gradient(x, y, beta):
    return np.dot(x.T,(y-sigmoid(x, beta)))

## Code based on https://stackoverflow.com/questions/38157972/
↪how-to-implement-mini-batch-gradient-descent-in-python
def create_batch(x,y,batch_size, shuffle=False):
    assert x.shape[0] == y.shape[0]
    if shuffle:
        indices = np.arange(x.shape[0])
        np.random.shuffle(indices)
    for start_idx in range(0, x.shape[0] - batch_size + 1, batch_size):
        if shuffle:
            excerpt = indices[start_idx:start_idx + batch_size]
        else:
            excerpt = slice(start_idx, start_idx + batch_size)
        yield x[excerpt], y[excerpt]

def mini_batch_GA(x, y, batch_size=42, alpha = 0.01, imax = 1000, precision = 0.
↪00000001):
    beta = np.zeros((x.shape[1], 1))
    for i in range(0,imax):
        mini_batches = create_batch(x, y, batch_size)
        for mini_batch in mini_batches:
            x_mini, y_mini = mini_batch
            beta = beta + alpha * gradient(x_mini, y_mini, beta)
    return beta

def log_likelihood(x,y,beta):
    return (y * betaX - np.log(1 + np.exp(betaX))).sum()

def AIC(x,y,beta,p):
    return 2*p - 2*log_likelihood(x,y,beta)
```

```
def log_loss(y, p):
    return -(y * np.log(p) + (1 - y) * np.log(1 - p)).mean()

def backward_search(x,y, xTest, yTest):
    v_used = [[i] for i in range(x.shape[1])]
    improvement = True
    iterations = 0
    AIClist = []
    loglosslist = []
    while(improvement):
        iterations += 1
        AIClist.append(AIC(x,y,mini_batch_GA(x,y),len(v_used)))
        gain_best = 0
        v_best = []
        for v in v_used:
            ## Calculate AIC for x without v
            x_without_v = np.delete(x, v, 1)
            beta_without_v = mini_batch_GA(x_without_v,y)
            AIC_without_v = AIC(x_without_v,y,beta_without_v, len(v_used))
            ## Calculate AIC for x with v
            x_with_v = np.insert(x, v, 1, axis=1)
            beta_with_v = mini_batch_GA(x_with_v,y)
            AIC_with_v = AIC(x_with_v,y,beta_with_v, len(v_used))
            gain = AIC_without_feature - AIC_with_feature
            v_used_v = [i[0] for i in v_used if i != v]
            if gain > gain_best:
                gain_best = gain
                v_best = v
        improvement = True if gain_best > 0 else False
        if improvement:
            v_used = [i for i in v_used if i != v_best]
        AIClist.append(gain_best)
        predictiontest = sigmoid(np.dot(beta, xTest.T))
        loglosslist.append(log_loss(yTest, predictiontest))
        finalerror = loglosslist[-1]
    return iterations, AIClist, finalerror

iterations, AIClist, finalerror = backward_search(x_train,y_train,x_test,y_test)
print(finalerror)

plt.plot(iterations, AIClist, label = "iteration vs AIC")
plt.grid()
plt.legend()
```

2 EXERCISE 2

2.1 Regularization for Logistic Regression

[ ]:
```
## Pick a range of  0 and   defined on grid
## You can choose fixed batchsize = 50.
## Implement k-fold cross-validation protocol for grid search
## For each combination of  0 and   you will perform k-fold cross-validation. Let k = 5 in this case
## Keep track of mean performance (i.e. Classification Accuracy value) across k - folds for each set of hyperparameters
## Plot on the grid  0 vs   the Classification Accuracy score for all combinations
## For the optimal value of alpha0 and  , train your model on complete training data and evaluate on Test data
## Report one single Accuracy and Log-likelihood for Test data
## Plot Train and Validation Accuracy and Log-likelihood metrics per k - fold iteration

imax = 100
alpha_list = np.array([0.01, 0.1, 1])
Lambda_list = np.array([0.1, 1, 10])
batch_size = 50
k = 5

def logreg_kfold(x, y, xTest, yTest, imax, k, alpha_list, Lambda_list, batch_size):
    beta = np.zeros(x.shape[1])
    ## Y prediction with lambda
    y_hat = lambda x,beta : (np.exp(x@beta)/(np.exp(x@beta)+1)).reshape(-1,1)
    def sigmoid(x,beta):
        return 1.0/(1.0 + np.exp(-1*(np.dot(x,beta))))
    def gradient(x, y, beta):
        return np.dot(x.T,(y-sigmoid(x, beta)))
    def log_likelihood(x,y,beta):
        return (y * betaX - np.log(1 + np.exp(betaX))).sum()
    def create_batch(x,y,batch_size, shuffle=False):
        assert x.shape[0] == y.shape[0]
        if shuffle:
            indices = np.arange(x.shape[0])
            np.random.shuffle(indices)
        for start_idx in range(0, x.shape[0] - batch_size + 1, batch_size):
            if shuffle:
                excerpt = indices[start_idx:start_idx + batch_size]
            else:
                excerpt = slice(start_idx, start_idx + batch_size)
        yield x[excerpt], y[excerpt]
```

```python
def mini_batch_GA(x, y, batch_size=42, alpha = 0.01, imax = 1000, precision = 0.00000001):
    beta = np.zeros((x.shape[1], 1))
    for i in range(0,imax):
        mini_batches = create_batch(x, y, batch_size)
        for mini_batch in mini_batches:
            x_mini, y_mini = mini_batch
            beta = beta + alpha * gradient(x_mini, y_mini, beta)
    return beta
## Grid
Grid = np.zeros((len(alpha_list)*len(Lambda_list),k))
## K-folds setting
index = np.random.permutation(x.index)
for i in range(1, k+1):
    x_val = x.loc[index][(i-1)* int(np.floor(x.shape[0]/k)):i* int(np.floor(x.shape[0]/k))]
    x_train = x.drop(index = x_val.index)
    y_val = y.loc[x_val.index]
    y_train = y.drop(index = x_val.index)
    Grid_counter = 0
    ## Alpha Loop
    for alpha_counter in range(len(alpha_list)):
        alpha = alpha_list[alpha_counter]
        ## Lambda Loop
        for Lambda_counter in range(len(Lambda_list)):
            Lambda = Lambda_list[Lambda_counter]
            ## Error vectors setting
            error_train = np.zeros(imax)
            error_val = np.zeros(imax)
            error_test = np.zeros(imax)
            ## Iterations
            for j in range(1,imax+1):
                ## Logistic regression setup
                index = np.random.permutation(x_train.index)
                ## Iterations of log reg
                for u in range(1, int(np.ceil(len(x_train)/batch_size))+1):
                    x_train_i = x_train.loc[Index][(i-1)*batch_size:i*batch_size]
                    y_train_i = y_train.loc[Index][(i-1)*batch_size:i*batch_size]
                    beta = mini_batch_GA(x_train_i,y_train_i)
                ## Log likelihood computation
                error_train[j-1] = log_likelihood(x_train,y_train,beta)
                error_val[j-1] = log_likelihood(x_val,y_val,beta)
            Grid[Grid_counter,i-1] = error_val[imax-1]
            Grid_counter += 1
```

```
logreg_kfold(x_train, x_test, y_train, y_test, imax, k, alpha_list,␣
↪Lambda_list, batch_size)
```

3 EXERCISE 3

3.1 Implementing hyperband

[82]:
```
## Firstly, re-split the dataset into train/validation/test splits according to␣
↪ratios: 70% : 15% : 15%

bank_num_train = bank_numeric.sample(frac=0.7,random_state=42) ## random state␣
↪is just a seed value
bank_num_leftover = bank_numeric.drop(bank_num_train.index)
bank_num_validation = bank_num_leftover.sample(frac=0.5,random_state=42)
bank_num_test = bank_num_leftover.drop(bank_num_validation.index)

print(bank_num_train.shape,bank_num_validation.shape,bank_num_test.shape)
```

```
(728, 41) (156, 41) (156, 41)
```

[ ]:
```
## Implement hyperband algorithm

## Code taken from https://homes.cs.washington.edu/~jamieson/hyperband.html

max_iter = 100  ## maximum iterations/epochs per configuration
eta = 3 ## defines downsampling rate (default=3)
logeta = lambda x: log(x)/log(eta)
s_max = int(logeta(max_iter))  ## number of unique executions of Successive␣
↪Halving (minus one)
B = (s_max+1)*max_iter  ## total number of iterations (without reuse) per␣
↪execution of Succesive Halving (n,r)

## Begin Finite Horizon Hyperband outlerloop. Repeat indefinitely.
for s in reversed(range(s_max+1)):
    n = int(ceil(int(B/max_iter/(s+1))*eta**s)) ## initial number of␣
↪configurations
    r = max_iter*eta**(-s) ## initial number of iterations to run␣
↪configurations for

    ## Begin Finite Horizon Successive Halving with (n,r)
    T = [ get_random_hyperparameter_configuration() for i in range(n) ]
    for i in range(s+1):
        ## Run each of the n_i configs for r_i iterations and keep best n_i/eta
        n_i = n*eta**(-i)
        r_i = r*eta**(i)
        val_losses = [␣
↪run_then_return_val_loss(num_iters=r_i,hyperparameters=t) for t in T ]
```

```
        T = [ T[i] for i in argsort(val_losses)[0:int( n_i/eta )] ]
    ## End Finite Horizon Successive Halving with (n,r)
```