Machine Learning 2016
homework 2 – Spam Classification

翁丞世, R04945028, mob5566[at]gmail.com

Logistic Regression Function

In this work, I modulize the logistic regression function as a class *logreg* in logreg_model.py. This section roughly describes how I implement the logistic regression. I will show the *fit* function of class *logreg* which can be used to train the input X with labels y with logistic regression.

I revise loss function of my linear regression from homework 1 to cross-entropy loss function as Prof. Lee taught on the class, and use stochastic gradient descent, Adagrad, and L2 regularization to train my logistic regression model.

```
66    def fit(self, X, y):
...
81        # use feature scaling
82        if self.useFS:
83            self.xmean = X.mean(axis=0)
84            self.xstd = X.std(axis=0)
85            X = (X-self.xmean)/(self.xstd+eps)
86
87        # data size
88        data_num, feat_num = X.shape
...
93        # initialize weights w and bias b with zeros
94        self._w = np.random.rand(feat_num)
95        self._b = np.array(0)
96
97        # accumulate delta
98        self.acc_dw = np.zeros(feat_num)
99        self.acc_db = np.array(0)
...
```

Here is ***fit(X, y)*** function of class *logreg* from loreg_model.py line 66 for training the input features X and labels y.

Feature scaling scales the input features to zero mean and unit variance.

Give random values to initialize the weights and bias.

Initialize the variables for Adagrad.

```
103         # gradient descent
104         for i in np.arange(self.maxIter):
105
106             if self.useSGD:
107                 rmask = np.arange(data_num)
108                 np.random.shuffle(rmask)
109                 mX = X[rmask]
110                 my = y[rmask]
111
112                 for i in np.arange(0, data_num-self.batchSize,
self.batchSize):
113                     tX = mX[i:i+self.batchSize]
114                     ty = my[i:i+self.batchSize]
115                     self.gradientDescent(tX, ty)
116
117             else:
118                 self.gradientDescent(X, y)
```

Train the model with input data by gradient descent *maxIter* times.

In this logistic regression, you can enable the *useSGD* to do stochastic gradient descent with specific *batchSize* for training your model.

Or you can just train your model with the whole batch input data.

```
127     def gradientDescent(self, X, y):
128         # calculate the gradient of cross-entropy with
current w and b
129         dw = np.negative(np.dot((y-sigmoid(np.dot(X,
self._w)+self._b)), X))
130         db = np.negative((y-sigmoid(np.dot(X,
self._w)+self._b)).sum())
131
132         # if use L2 Regularization
133         if self.useL2R:
134             dw = dw+2*self.L2R_lambda*self._w
135
136         # if use Adagrad
137         if self.useAdagrad:
138             self.acc_dw = self.acc_dw+dw**2
139             self.acc_db = self.acc_db+db**2
140
141             dw = dw/np.sqrt(self.acc_dw+eps)
142             db = db/np.sqrt(self.acc_db+eps)
144         # update the w and b
145         self._w = self._w-self.eta*dw
146         self._b = self._b-self.eta*db
```

This *gradientDescent* function calculates the gradient of current model.

$$z = Xw + b$$

$$\nabla w = -\Big(\big(y - \sigma(z)\big) \cdot X\Big)$$

$$\nabla b = -\sum y - \sigma(z)$$

$$\sigma(z) = \frac{1}{1 + e^{-z}}$$

You also can use L2 Regularization to add an L2 regularizer behind the square error.

Adagrad can auto adjust the learning rate in gradient descent.

Finally update the current weights and bias with calculated gradient.

```
204 def sigmoid(X):
205     signal = np.clip(X, -500, 500)
206     signal = 1.0/(1 + np.exp(-signal))
207     return signal
```

The sigmoid function is first clip the input data values into the interval [-500, 500] for solving floating point precision overflow problem.

Another Method - Random Forest

I choose the random forest for classification as my second method. Actually, I had implemented the random forest for regression in homework 1, but the performance was unfavorable. In this work, I implemented the random forest model as *decision_tree.py* and *ensemble_model.py*.

In each tree node of decision tree, I randomly select some features to find the best split with minimum impurity of the present data. Then I can branch the node with the best split to next tree node recursively. Therefore, I bootstrap the data to construct trees to become a forest with specified size. The random forest can overcome the overfitting problem of decision tree.

```
 55     def fit(self, X, y):
...
 59         # generate random feature sampling mask
 60         mask = np.arange(feat_num)
 61         np.random.shuffle(mask)
 62
 63         if self.max_features and self.max_features<feat_num:
 64             mask = mask[:self.max_features]
 65
 66         # terminate
 67         if impurity(y) < self.min_impurity or \
 68             np.all(X[:, mask].std(axis=0)<self.min_feat) or \
 69             (self.max_depth<1 if self.max_depth!=None else
False) or \
 70             len(X)==1:
 71
 72             self.isLeaf = True
 73             self.val = 1 if y.mean()>0.5 else 0
 74             return self
```

The ***fix(X, y)*** function is the same API in decision tree as in logistic regression.

I implement the random feature space transform which choose specific number of features when branching in decision tree.

Check the criterion of termination of decision tree. 1. The impurity is too small. 2. The features are almost identical. 3. There is specific tree depth 4. The data size is 1.

If the criterion is met, this tree is set to a leaf node.

```
 79        for i in mask:
 80            interv = np.array(list(set(X[:,i])))
...
 83            interv = np.array([(interv[k]+interv[k+1])/2 \
 84                for k in np.arange(len(interv)-1)])
 85
 86            for val in interv:
 87                smask = X[:, i]<val
 88
 89                imp = impurity(y[smask])*smask.sum() +\
 90 impurity(y[np.logical_not(smask)])*np.logical_not(smask).sum()
 91
 92                if imp < minImp:
 93                    minImp = imp
 94                    self.split_feat = i
 95                    self.split_val = val
 96                    minSmask = smask
 97
 98        if minSmask==None or np.all(minSmask):
 99            self.isLeaf = True
100            self.val = 1 if y.mean()>0.5 else 0
101            return self
102
103        nextdep = self.max_depth-1 if self.max_depth else
None
104        self.childs = [\
105            dtree(nextdep, self.max_features, self.min_feat,
self.min_impurity),\
106            dtree(nextdep, self.max_features, self.min_feat,
self.min_impurity)]
107
108        self.childs[0].fit(X[minSmask], y[minSmask])
109        self.childs[1].fit(X[np.logical_not(minSmask)],
y[np.logical_not(minSmask)])
110
111        return self
```

Find out the split in one of the chosen feature with the smallest impurity.

Compute the avalible intervals in chosen feature.

Compute the impurity of each split.

Record the split with minimum impurity.

If there is no split, terminate the tree.

Branch the data with seltected split, and build the child trees recursively.

```
 83     def fit(self, X, y):
 84         self.trees = []
 85 
 86         sampleNum = self.sampleNum if self.sampleNum else
len(X)
...
 91         self.oob_score = 0
 92 
 93         # for each random trained trees
 94         for i in np.arange(self.treeNum):
 95 
 96             # random data sampling
 97             mask = np.random.randint(0, len(X), sampleNum)
 98 
 99             # get the out-of-box validation set
100             oobmask = np.ones(len(X), dtype=bool)
101             oobmask[mask] = False
102 
103             # add the tree
104             self.trees.append( dt.dtree(self.max_depth,
self.max_features,\
105                                 self.min_feature,
self.min_impurity) )
106 
107             # train teh decision tree
108             self.trees[-1].fit(X[mask], y[mask])
109 
110             # calculate this decision tree score
111             self.oob_score = self.oob_score+\
112                 self.scoring(self.trees[-1], X[oobmask],
y[oobmask])
113 
114         self.oob_score = self.oob_score/self.treeNum
```

The ***fix(X, y)*** function from class *random_forest* in *ensemble_model.py*.

Bootstrap specific number of data to build a forest.

Add and train a decision tree from a bootstraped data.

Compute the Out-of-Box score of this decesion tree.

Obtain the Out-of-Box score of this random forest by averaging the decisiont tree scores.

Discussion

Logistic Regression

I had implemented the validation and cross-validation in *model_selection.py* and used 10-fold cross-validation to estimate my logistic regression models.

First, I tested the models with different learning rates, and the result showed that the learning rate 0.05 had the best learning curve with fast convergence and stability. After that, I chose the model with minimum error by cross-validation from different L2 regularizer. Finally, I got my best logistic regression model with learning rate 0.05, L2 regularizer 0.03, and trained 2000 iterations.

As a result, my best logistic regression model performed 0.927 accuracy in in-sample error and 0.924 accuracy in cross-validation with training around 20 seconds.

Random Forest

In the random forest, I implemented the out-of-bag score evaluation, so I can choose the best random forest with different parameters (e.g. number of features sampling, depth of decision tree, etc.) by out-of-bag score.

I tried different number of features sampling, and I find that the maximum out-of-bag score was achieve when I use 25 random features to split the data in each tree node. In my opinion, the appropriate feature number can increase the randomness of random forest, and also reduce the over-fitting problem. I also tried different depth of tree. I thought if the depth of tree decrease, then the complexity of decision tree would decrease, and the over-fitting problem could also be decrease. Therefore, I chose the best random forest with 20 tree depth and 25 random features sampling.

I wanted the better accuracy, so the number of trees in random forest should be as much as possible in the time limit. In my case, training a decision tree with 20 tree depth and 25 random features sampling cost around 7 seconds. In kaggle, I submitted the results with 600 trees in random forest. In Github, I submitted the model with 100 trees in random forest.

My random forest with 100 trees, 20 tree depth, and 25 random features sampling performed 0.965 accuracy in out-of-bag score. Obviously, the random forest model outperformed the logistic regression model.