Machine Learning 2016
homework 1 - P.M. 2.5 prediction

翁丞世, R04945028, mob5566[at]gmail.com

Linear Regression function by Gradient Descent

I modulize the linear regression function as a class *linreg* in linreg_model.py. This section roughly describes how I implement the linear regression.

(Some code blocks for variables initialization, parameter assignment would be omitted by '…' for readability.)

Here is ***fit(X, y)*** function of class *linreg* from linreg_model.py line 100 for training the input features *X* and labels *y*.

```
100    def fit(self, X, y):
...
115        # use feature order transform
116        if self.featureOrder:
117            tX = X
118            for i in np.arange(2, self.featureOrder+1):
119                tX = np.append(tX, X**i, axis=1)
120            X = tX
121
122        # use feature scaling
123        if self.useFS:
124            self.xmean = X.mean(axis=0)
125            self.xstd = X.std(axis=0)
126            X = (X-self.xmean)/(self.xstd+eps)
...
134
135        # accumulate delta
136        self.acc_dw = np.zeros(X.shape[1])
137        self.acc_db = np.array(0)
```

Feature transformation transform the input features to higher order feature space.

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

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

In this linear 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.

```
        # gradient descent
        for i in np.arange(self.maxIter):

            if self.useSGD:
                rmask = np.arange(len(X))
                np.random.shuffle(rmask)
                mX = X[rmask]
                my = y[rmask]

                for i in np.arange(0, len(X)-self.batchSize, self.batchSize):
                    tX = mX[i:i+self.batchSize]
                    ty = my[i:i+self.batchSize]
                    self.gradientDescent(tX, ty)

            else:
                self.gradientDescent(X, y)
```

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

$$\nabla w = 2 * (Xw + b - y) * X$$

$$\nabla b = 2 * (Xw + b - y)$$

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.

```
    def gradientDescent(self, X, y):
        # calculate the gradient of square error with current w and b
        dw = np.dot(2*((np.dot(X, self._w)+self._b)-y), X)
        db = 2*(((np.dot(X, self._w)+self._b)-y).sum())

        # if use L2 Regularization
        if self.useL2R:
            dw = dw+2*self.L2R_lambda*self._w

        # if use Adagrad
        if self.useAdagrad:
            self.acc_dw = self.acc_dw+dw**2
            self.acc_db = self.acc_db+db**2

            dw = dw/np.sqrt(self.acc_dw+eps)
            db = db/np.sqrt(self.acc_db+eps)

        # update the w and b
        self._w = self._w-self.eta*dw
        self._b = self._b-self.eta*db
```

Method Description

Bellow I will describe my method with two parts **data preprocessing** and **training**.

Data preprocessing

In the train.csv, there are 18 features in each hour, and 24 hours a day, and 20 days a month. There are 480 continuous hours each month, so I extract every 10-continuous-hours from each month as one sample. Every example has 9-hours features (9*18) = 162 dimension and 10^{th} hour P.M. 2.5 as label. There will be (480-10+1) = 471 continuous-10-hours per month, so I have 471*12 = 5652 samples with 162 feature dimension and 1 label.

Training

After several linear regression models testing, I found that the in-sample error and validation error are both around 5.8. I think it may be under-fitting, so I transform the features into higher order (e.g. 2-order X^2, or higher). Once I transform the features to higher order space, the in-sample error would decrease, but the validation error would rise. It's over-fitting. I decide to reduce the dimension of original 162 features. I train a linear regression model the original 162 features, then I choose the top 50 features with higher weight absolute magnitude to be my new features. Consequently, it gets better performance in higher order feature transform.

Regularization

In this work, I find that the regularization is kind of useless to the model with lower model complexity. But when I use the model with non-linear feature transform, it will have better performance on validation error than the models without regularization.

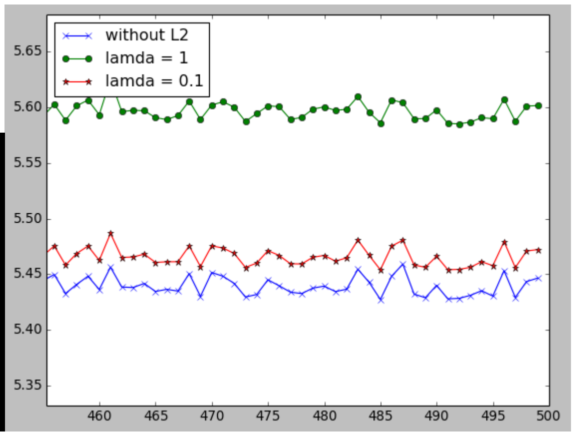
For an instance, here are three non-linear 4-order transform regression models trained by stochastic gradient descent iterating 500 times. The figure bellow is the iterating round and in-sample error.

We can find that the blue line without regularization has lowest in-sample error, the red line with smaller regularizer has larger in-sample error than blue line, but smaller than green line with larger regularizer.

The table next to the figure is errors of in-sample and validation from top to down, blue, green, red, respectively.

We can find that the larger regularizer performs better on validation error.

```
Ein:
[[ 5.44654922]
 [ 5.60133736]
 [ 5.47212318]]
Scores:
[[ 6.3045838 ]
 [ 6.23885159]
 [ 6.2453072 ]]
```

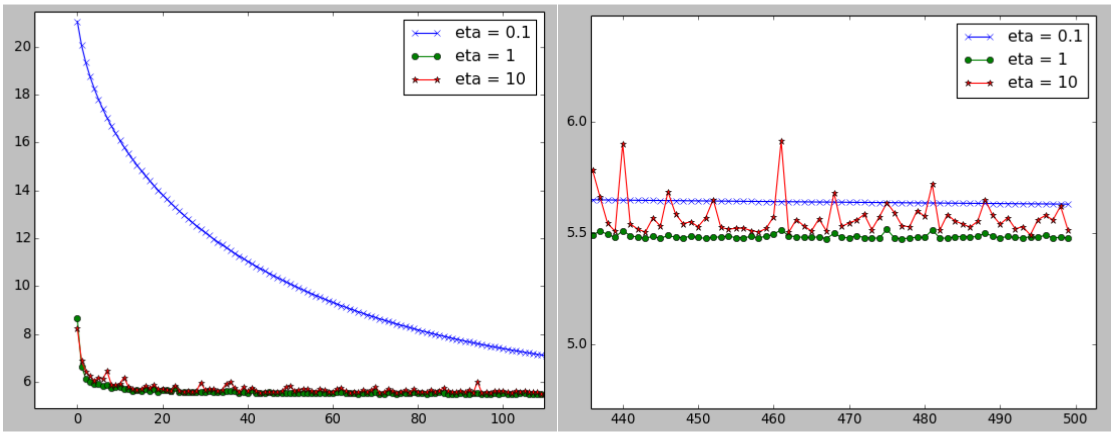


Learning Rate

When we use gradient descent, learning rate η(eta) is a critical factor of our model in training phrase. If we use larger learning rate, the error will decrease faster but unstable. On the other hand, if we use smaller learning rate, the error will decrease slowly but error curve smooth.

For an instance, here are three non-linear 2-order transform regression models trained by stochastic gradient descent iterating 500 times. The figures bellow are the iterating round and in-sample error plots.

The left one is in early training phrase from training start to iterating stochastic gradient descent 100 times, we can find that the smaller learning rate error, the blue line, is fall slowly and smoothly. But in the right figure which is the late training phrase from 440 rounds to end of training, the error with larger learning rate, the red line, is very unstable. Eventually, I choose the median one, the green line, which decrease fast and also stable.



Other Discussion

In fact, I also implement other models except linear regression. There are validation and cross-validation in *linreg_model.py* for model selection.

And decision tree model implemented by C&RT (Classification and Regression Tree) in *decision_tree.py* actually is prepared for random forest model.

And ensemble meta models, bagging model and random forest, in *ensemble_model.py* are expected to get better performance for under-fitting. After I implement the models and test on the data, it seems not working perfectly as I thought. The random forest performs perfect on training data, but very awful on testing. Due to lack of experience on Machine Learning, I don't realize why the powerful models do not work at the first week. I spent all my time for tuning the parameters of random forest first week. After doing survey on random forest, I think my problem is there are not enough trees in my forest. But I can't get more trees due to bad implementation of my random forest, it is time consuming. At the end, I give up the random forest and change direction to higher order feature space.

In this homework, I learn a lot about linear regression, random forest, and validation in practice .