1. Linear regression function by Gradient Descent

I implemented gradient descent in 3 different ways. Aside from the Adagrad mentioned in class, I also tried out Adadelta and RMSprop in attempt to ameliorate the deteriorating learning rate in late iterations. I obtained the details on these two methods via this website (sebastianruder.com/optimizing-gradient-descent).

However, after implementing all these three variations of gradient descent, Adagrad still turned out to be the best for this scenario. The other two would have a much slower start than Adagrad, and fail to drop to an acceptable amount of RMSE even after 0.5Mil of iterations

```
for it in range(it_max):
    est_mat = w0 + M_mat * w1_mat
    err_mat = Y_mat - est_mat
    if it%100==0:
        print "[",it,"/",it_max,"]\t",np.sqrt(np.sum(np.array(err_mat)**2)/5652)
    # Gradient Descent
    w0_grad = 2*np.sum(err_mat) # linear offset
    w1_grad = (-2*M_mat.T * err_mat) # linear weights
    # Regularization
    w1_grad = w1_grad + 2*lam*np.array(w1_mat)
    # Adagrad
    ada_w0 = np.sqrt(ada_w0**2 + w0_grad**2)
    ada_w1 = np.sqrt(np.square(ada_w1) + np.square(w1_grad))
    # Update Step
    w0 = w0 - rate*w0_grad/(ada_w0+eps)
    w1_mat = w1_mat - rate*w1_grad/(ada_w1+eps)
```

Fig. 1

In Fig.1 , the main gradient descent code is shown, as the variable names suggest, the operations are mainly done by matrix operations. How *M_mat* is constructed is described in the next section

2. Describe your method

Feature Selection

Since we're predicting the 10^{th} hour's PM2.5 based on the 9 previous hours of data, my selection of features is all the data in the 9 previous hours.

I first transformed the data into a matrix of size [18,5760], each row corresponds to the data of 1 measured attribute (ex: AMB_TEMP), and each column corresponds to the sampled time in each month.

Then the matrix is extracted 9 columns at a time to construct the set of features. For example, the first set of features consists of the first 9 columns in the matrix, which is a total of $18\times9 = 162$ data. Hence, every prediction is based on these 162 features.

Intuitively, we could extract $5760\text{-}9 = 5751$ set of features. However, the data given has a discontinuity between each month. For example, the 480^{th} column corresponds to the data measured in 1/20, yet the 481^{th} data is from 2/1.

Therefore, it might not be appropriate to just concatenate these feature points together.

I made **2 variations of feature selection**, the 1st one uses 5751 set of features, which ignores the discontinuity between months; the 2nd one uses 5652 set of features, which skips the first 9 samples in every month.

With the same settings(Linear model/ $\lambda = 1$), **the 2nd variation yields a slightly better result than the 1st**, as shown in the table below.

Variation	1st	2nd
RMSE	5.7478	5.7426

Models Used

Aside from linear models, I also tried using **quadratic and polynomial models**. However, the latter 2 models suffer from overfitting issues.

I also tried a **combination of classification and linear regression,** in which I train 12 different sets of linear weights correspond to each month. Then attempt to classify which month based on a given set of features. I implemented a naïve classification based on the 9 AMB_TEMP features, and find the best fit in all the data. Based on some experiments on training sets, its classification accuracy is still pretty poor. Alternatively, I also tried classifying 12 months into 4 seasons, which should be a lot more accurate than the former one. Yet, it only yields a ~50% prediction accuracy, and after submitting it to Kaggle, its public test's result were a lot worse than linear baseline, ~7 to 8 RMSE.

Implementation

During implementation, I used the **linear algebra solution** as the initial weights. This provides a much faster convergence during initial iterations. And since the linear model is convex, theoretically it wouldn't sabotage the result and fall into some local minimum. Besides, transforming all operations into matrix operations also speeds up the script. With this setup, the initial RMSE is already the minimum, then further relaxes the linear weights with respect to the regularization parameter (λ)

Additionally, I also tried **feature scaling**, in order to speed up the gradient descent. However, it doesn't seem to yield better results than using Adagrad solely based on some rudimentary trials.

Moreover, PM2.5 values should be non-negative integers (except for probably some measurement errors), so if the final output for the test set contains negative values, some **zero-forcing** could be done to slightly improve the RMSE.

---

3. Discussion on regularization

Regularization smooths the weights to avoid overfitting by adding a bias term to

the model. During this homework, I mostly experimented on lambda = 1, 100 and 1000.

Based on observations of my previous submissions on Kaggle, lambda=1000 yields the best result on the public set. A rudimentary analysis on how regularization affects the convergence is shown in the following table. Note that the RMSE is calculated based on the Kaggle public set.

Linear model, Rate = 1, Using Adagrad, 50000 iteration, Initial weight = zeros			
λ	1	100	1000
RMSE	5.72794	5.72710	5.72408

The parameters above are tuned to emphasize the difference of different λ settings, so they do not necessarily yield the best overall competition result.

We can see that as λ increases, the RMSE of predicting the public set slightly improves. However, when λ goes further than 1000, such as 1500, it usually starts deteriorating.

---

4. Discussion on learning rate

If the gradient descent is set correctly, preferably with Adagrad, then if the learning rate is set too high (ex: >10), the 1st iteration would overshoot the loss too high, and then gradually drop over the rest of the iterations. Hence, it would save a lot more time if the learning rate is set to a appropriate amount. On the other hand, if the rate is set too low, it wouldn't be able to converge to an appropriate result.

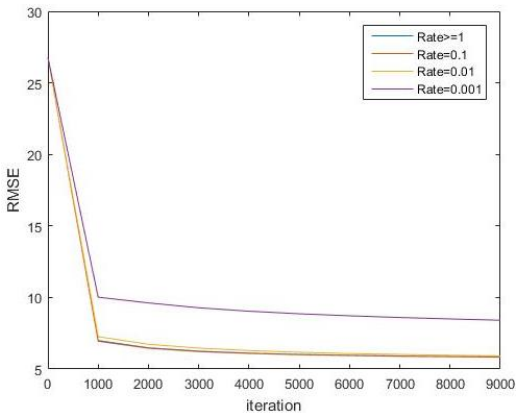

Fig. 2

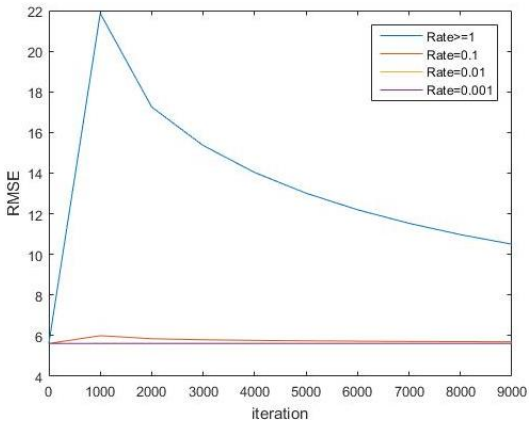

Fig. 3

Fig. 2 and Fig.3 are simulations done on linear model, using λ=1000, with Adagrad. Fig.2 uses 0 for all initial weights, and Fig.3 uses the least square error solution as initial weights. We can see that 0.1 and 0.01 is an appropriate choice for either case.

Notes on kaggle_best.py

kaggle_best.py is identical to linear_regression.py, since it yielded the best public set result. However, my best private set result is obtained from using a quadratic model, with $\lambda_1 = 100$ for linear weights and $\lambda_2 = 1000$ for quadratic weights,

which is trained through 2M iterations.