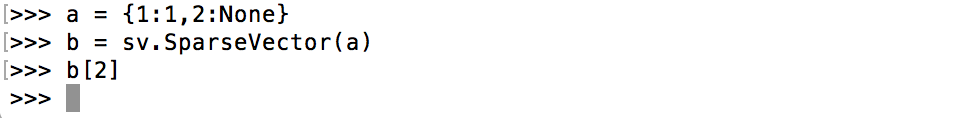
**HW 3**

**Xianlong Zhang**

**Question 1:**

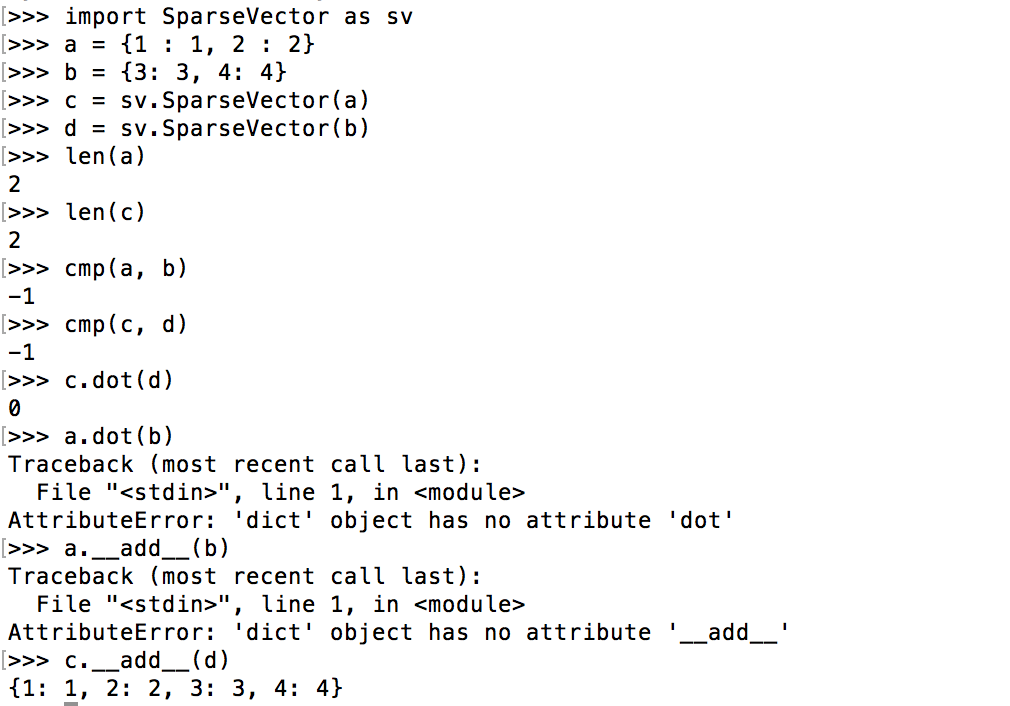
**(a)**

The class SparseVector is similar to a standard dictionary, but it would return nothing if a value of a key is None, however, a standard dictionary would return None.

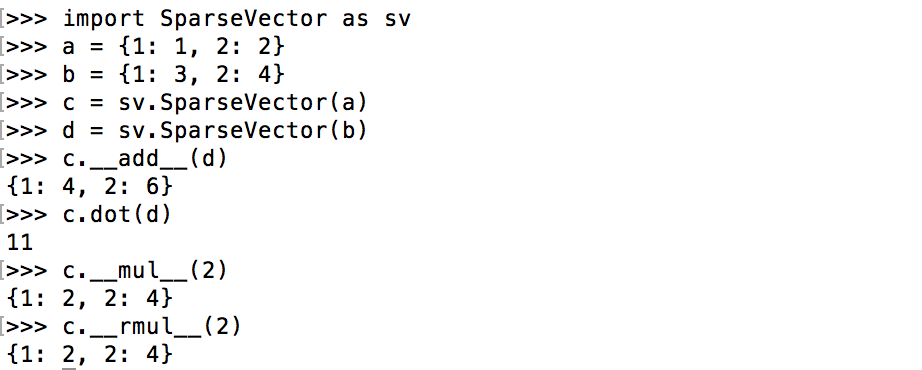
****

**(b)**

From the following examples, we can see that both standard dictionary and SparseVector have cmp() and len() methods, but SparseVector also has dot() and \_\_add\_\_() methods, which are not presented in dictionary.

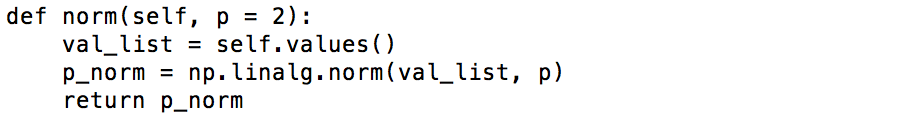
****

**(c)**

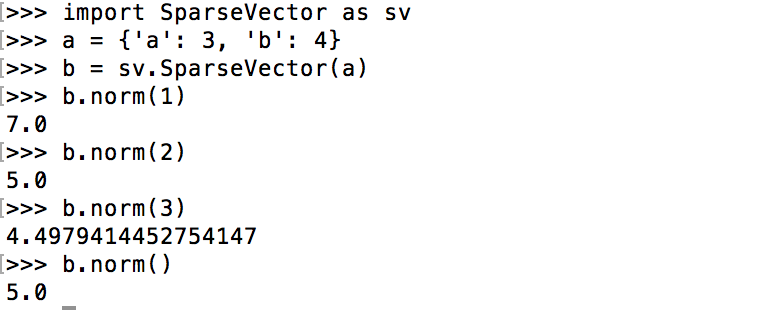
****

**(d)**

The modified code is as following:

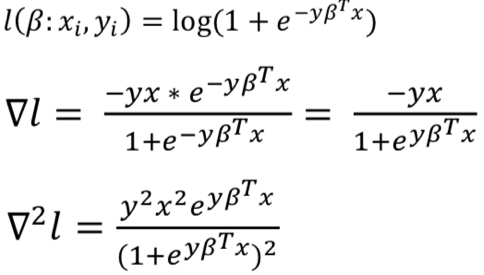


and for p = 1, 2, 3, the outputs are as following:

****

**Question 2:**

**(a)**

****

Since as well, so is a convex function.

**(b)**

As shown in (a), .

**(c)**

Since both and are associated with , once , , then they don’t depend on for multiply 0.

**(d)**

Since is convex, is also convex. Let , then and .So if , is convex.

**(e)**

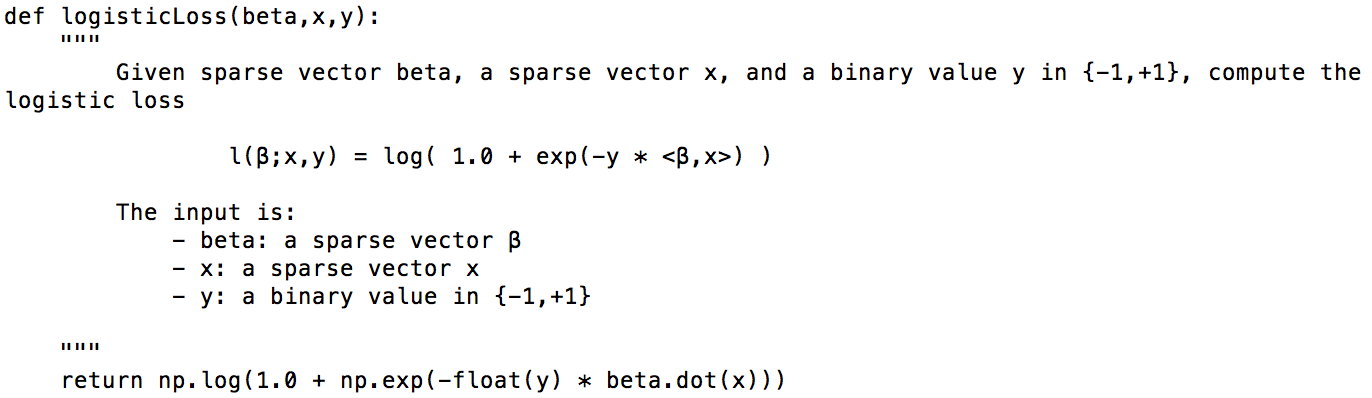
From (d) we know , only when , can L be a strictly convex function, so when, L is not strictly convex.

**(f)**

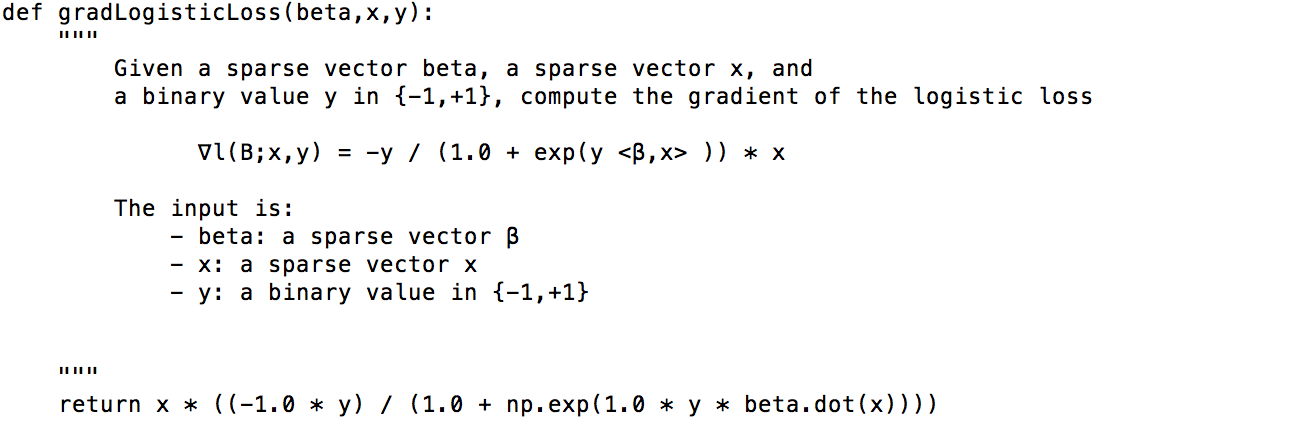
From (d) we know , only when , where , can L be a convex function, so when, , then L is strongly convex.

**Question 3:**

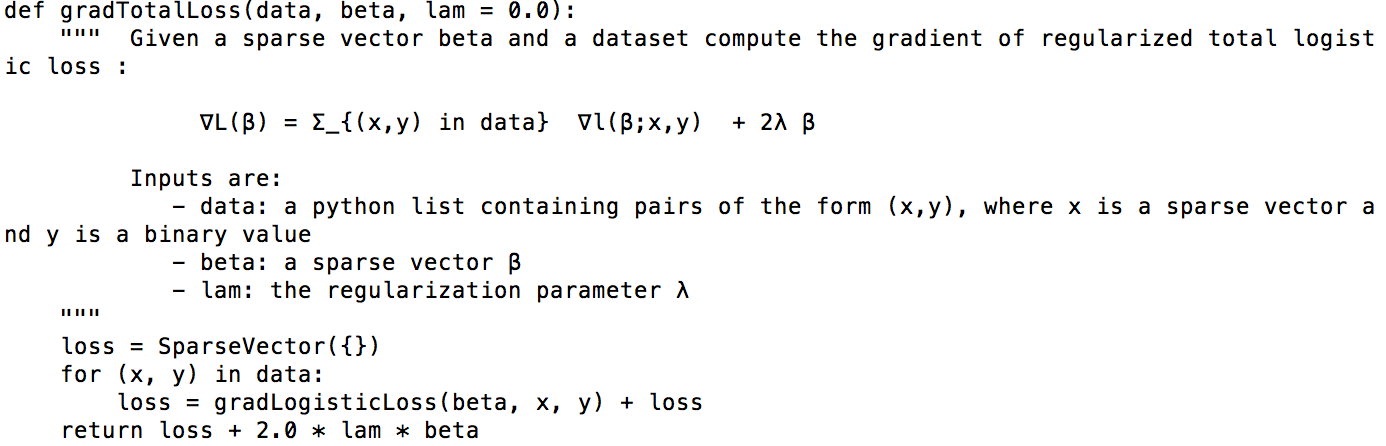
**(a)**

****

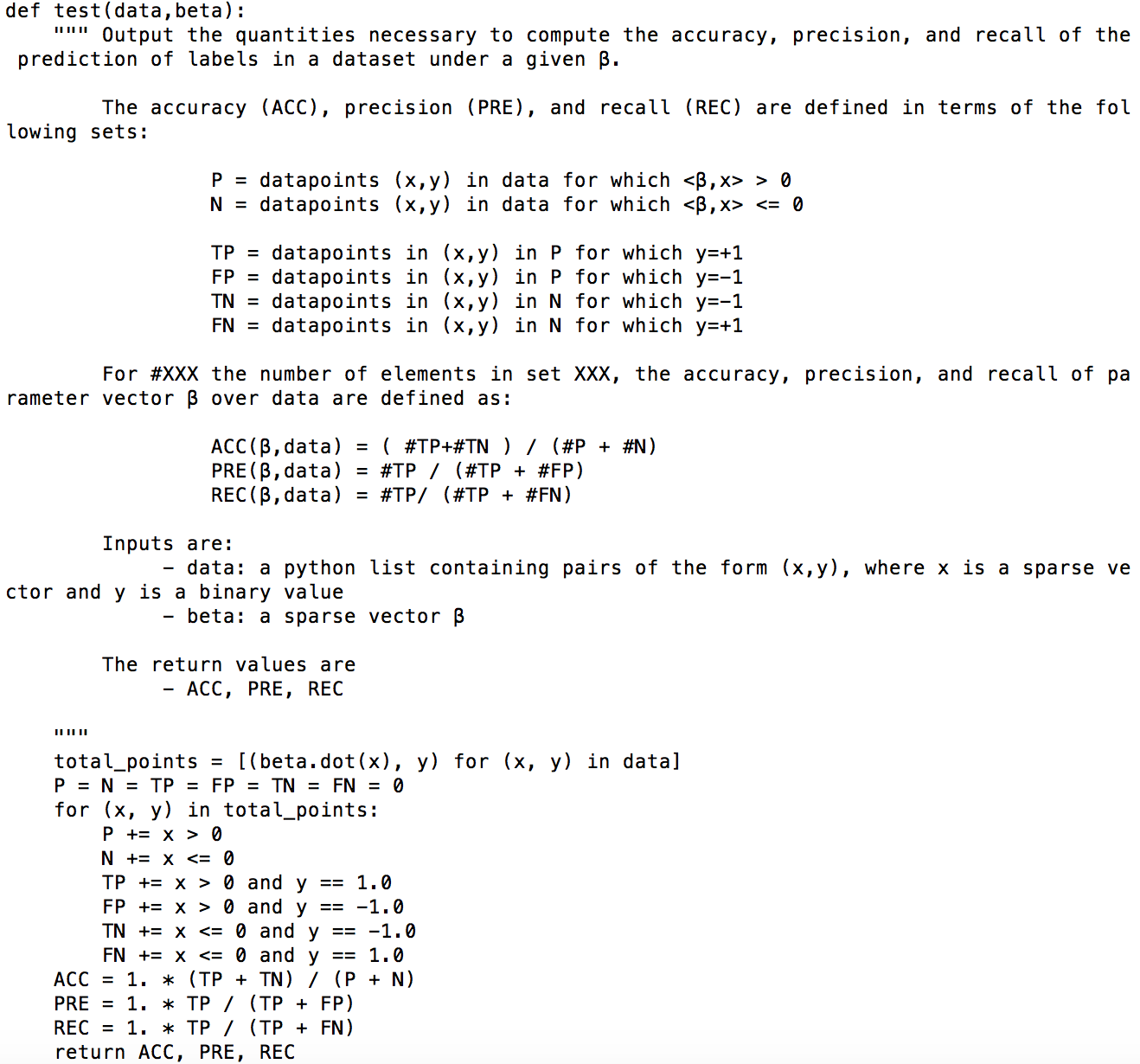
**--------------**

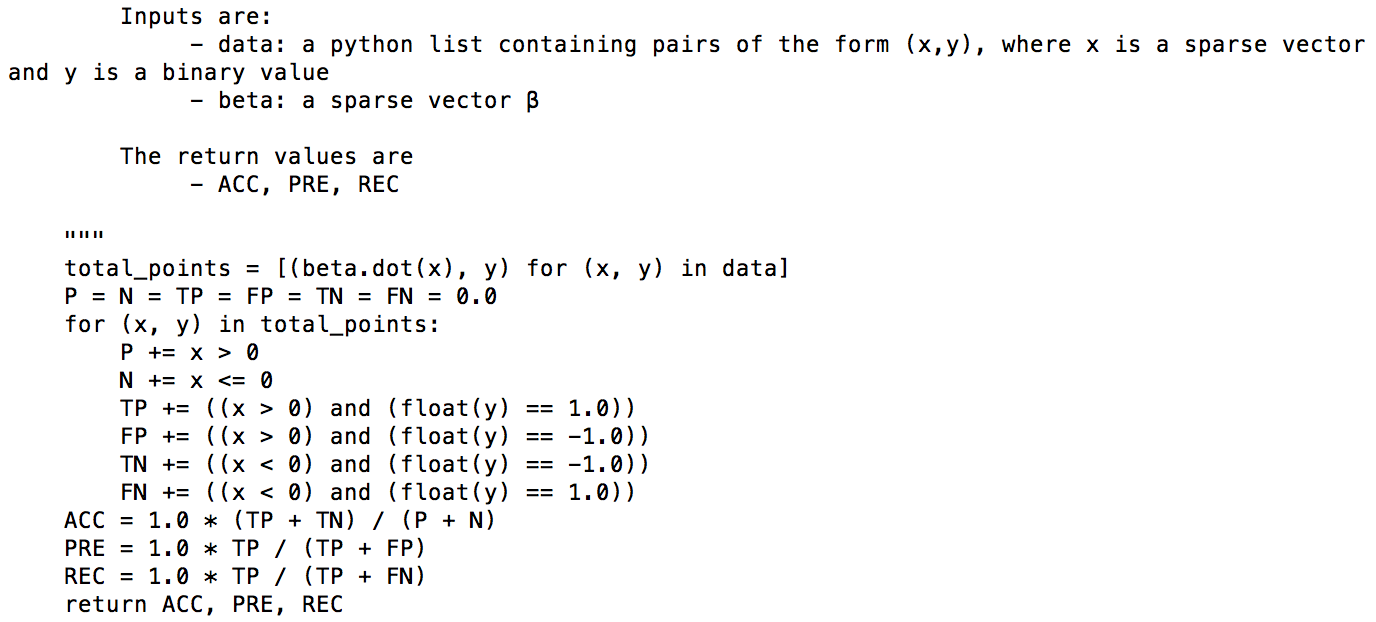
****

**--------------**

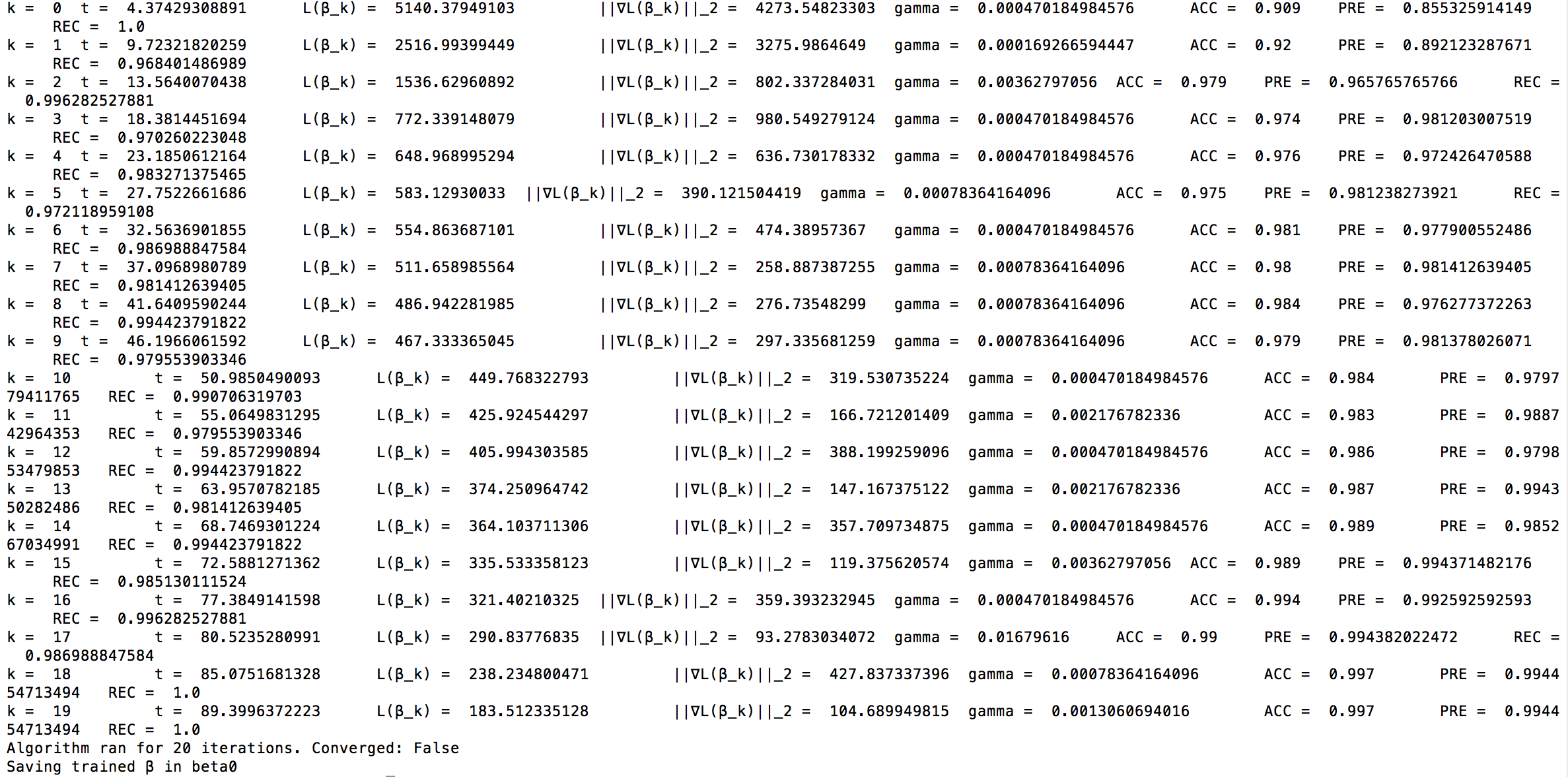
****

**(b)**

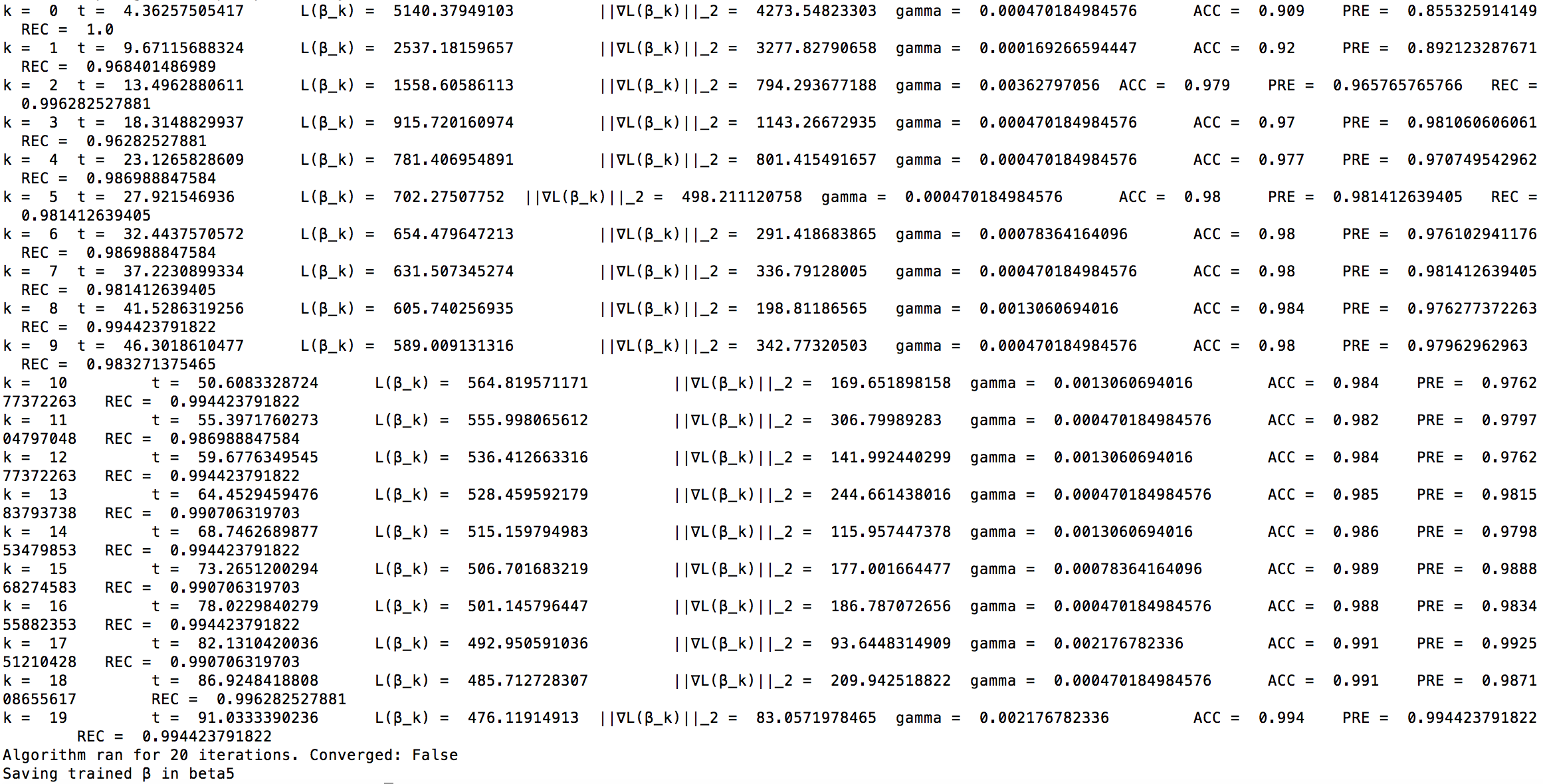
****

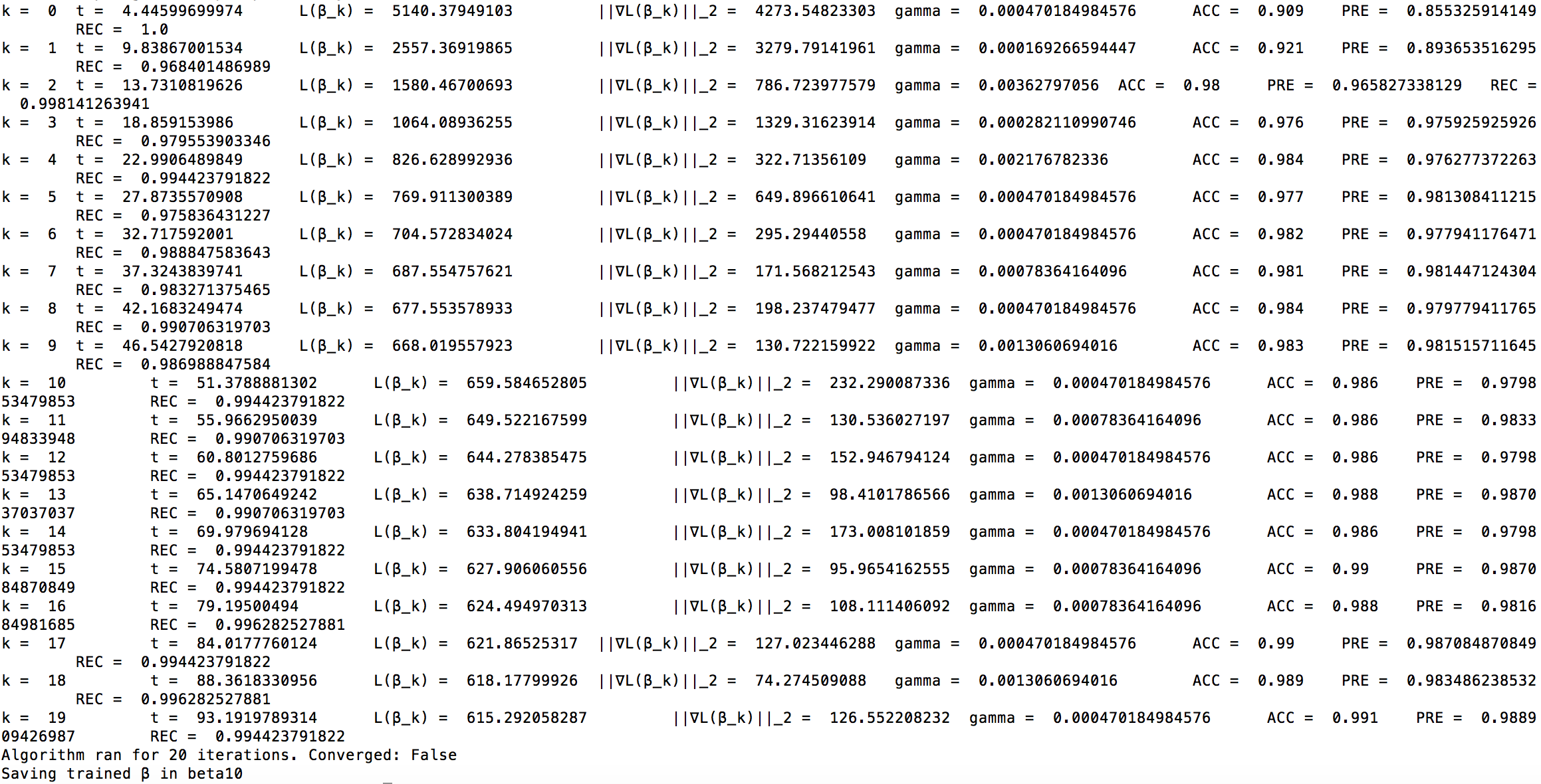
****

**(c)**

**** When ,

When ,



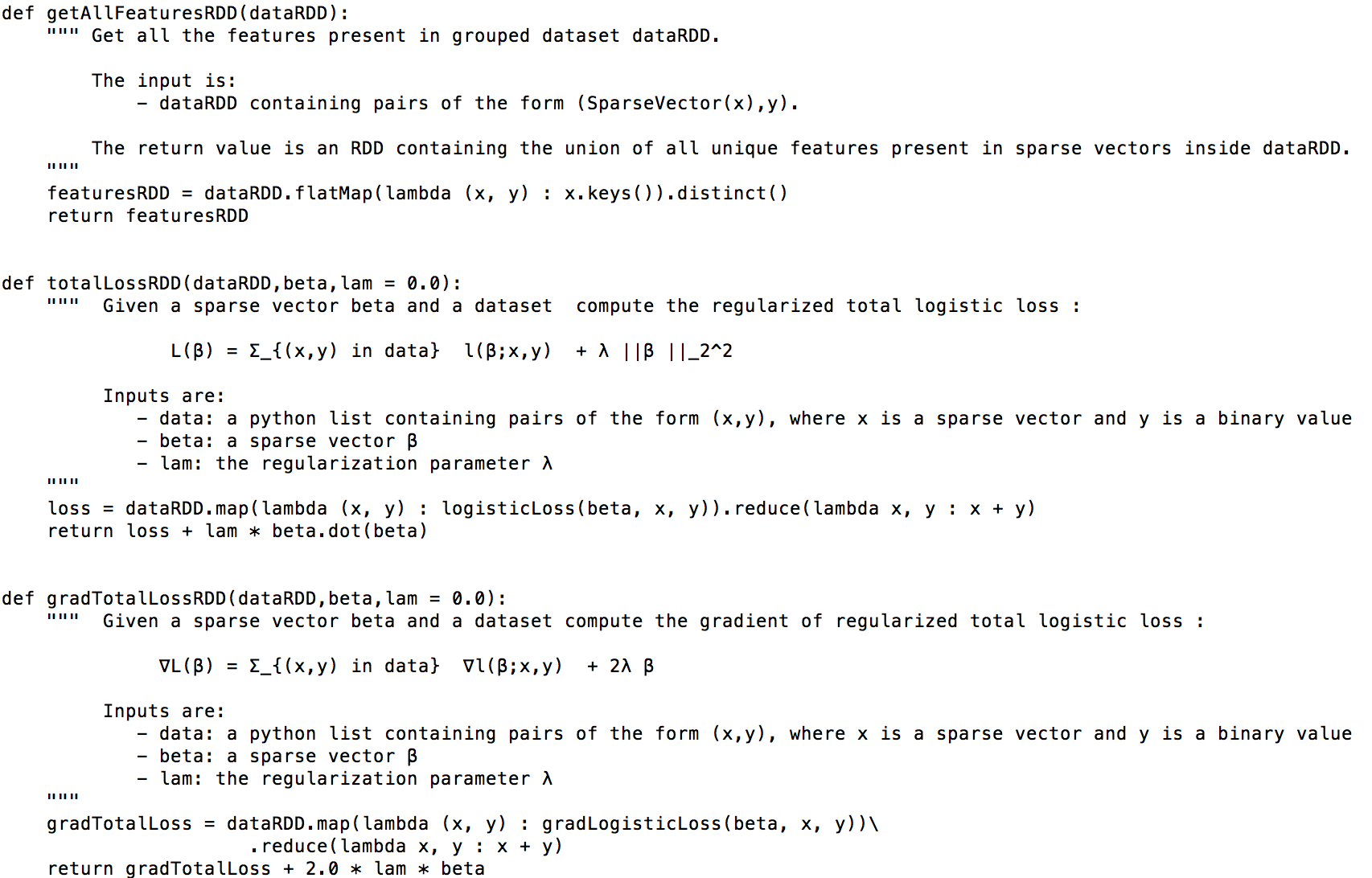
When ,

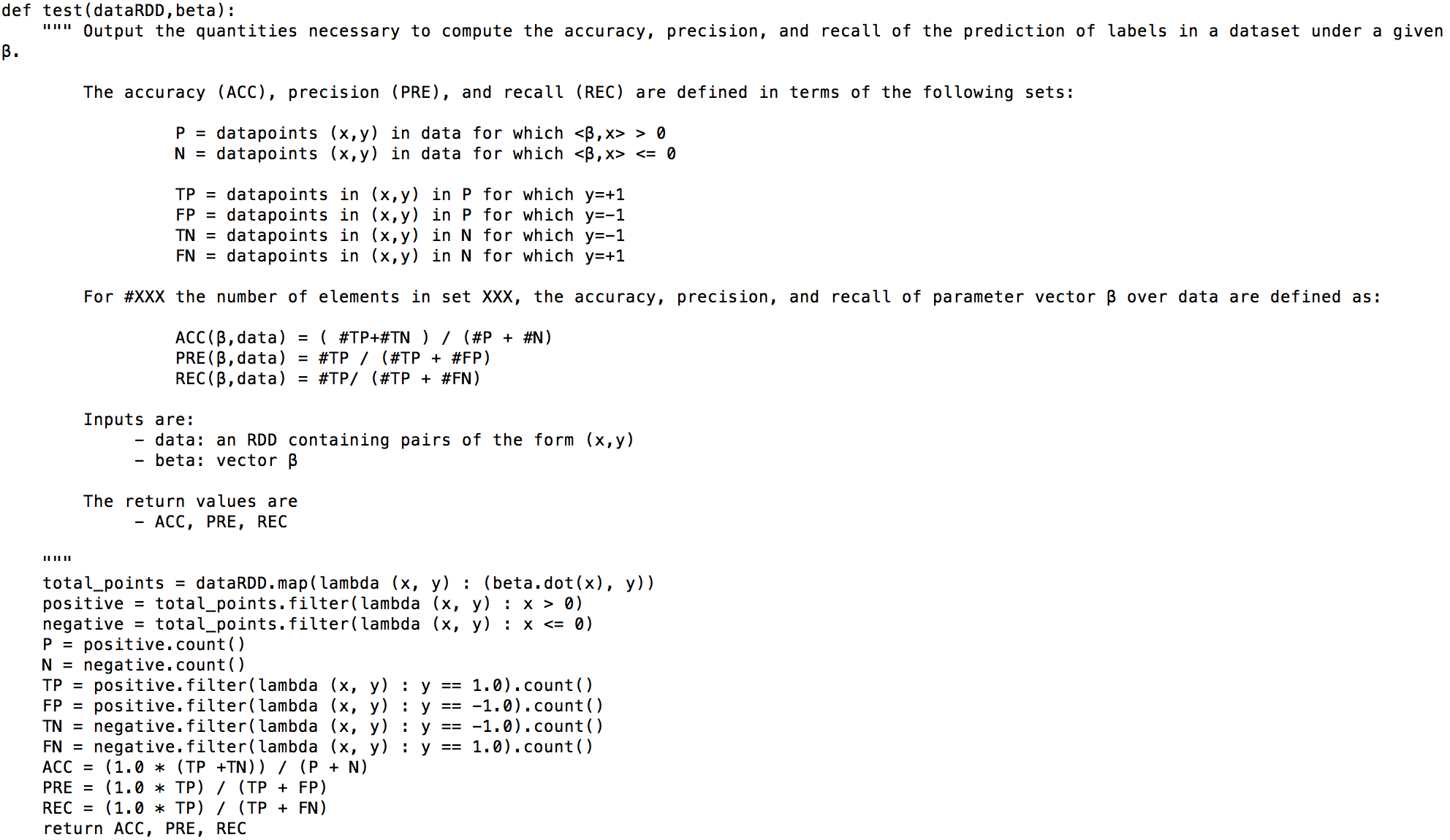
**(d)**

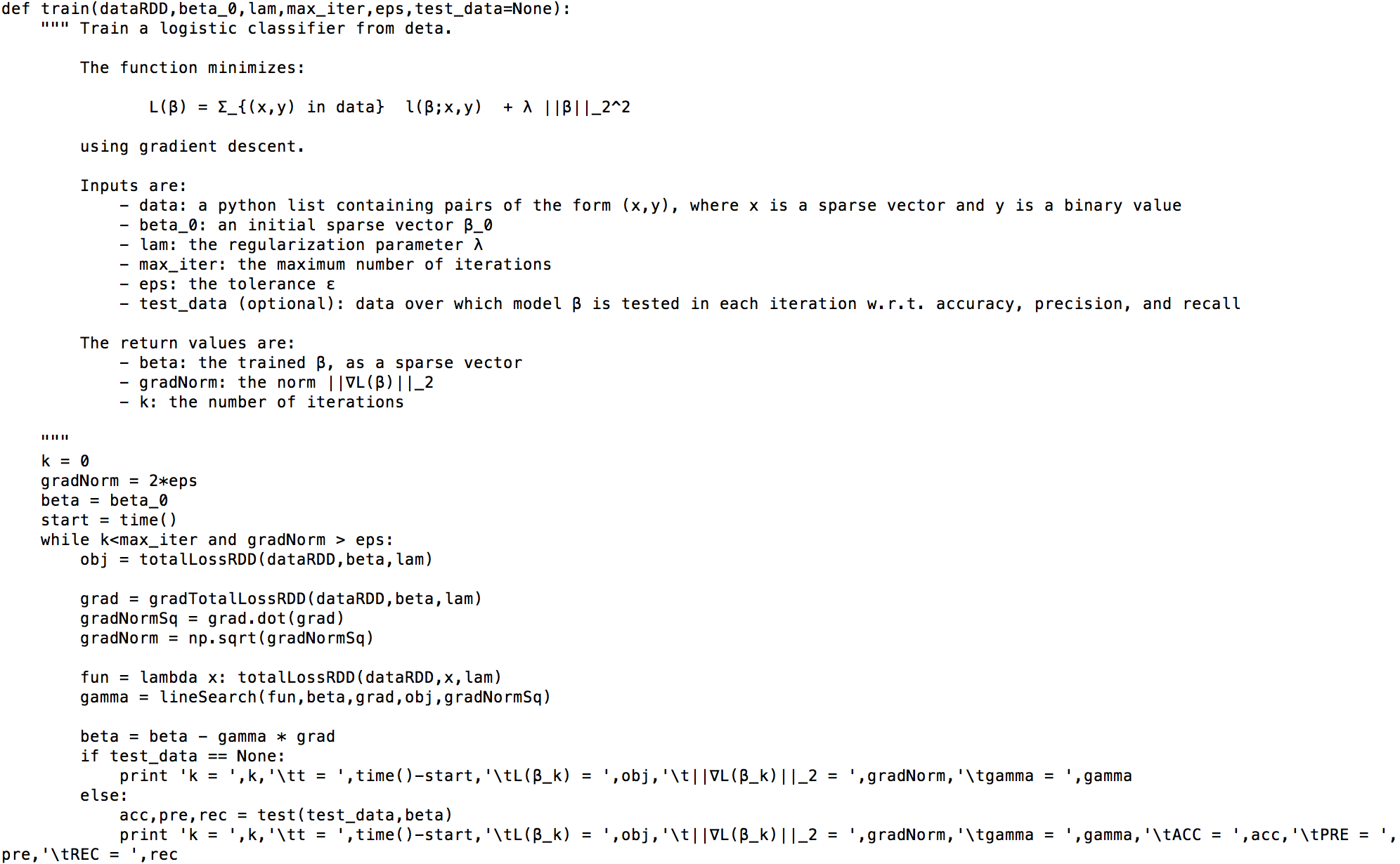
Recall is TP/(TP + FN) whereas precision is TP/(TP+FP). Recall means the fraction of relevant instances retrieved over total number of relevant instances, and precision means the fraction of relevant instances retrieved among all relevant instances. In this case, since the mushroom maybe poisonous, and harmful to healthy, we prefer high precision and low recall.

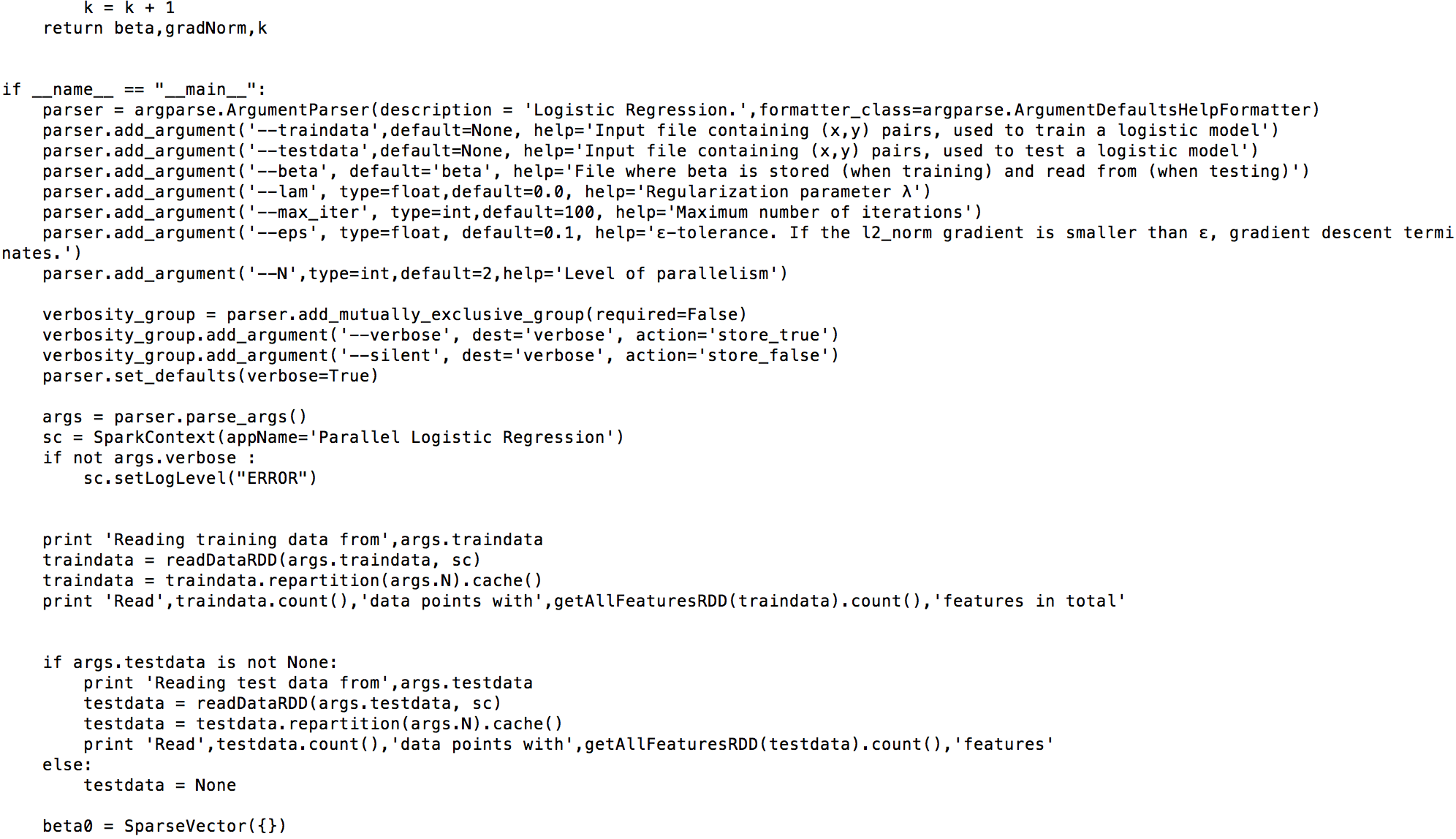
**Question 4:**

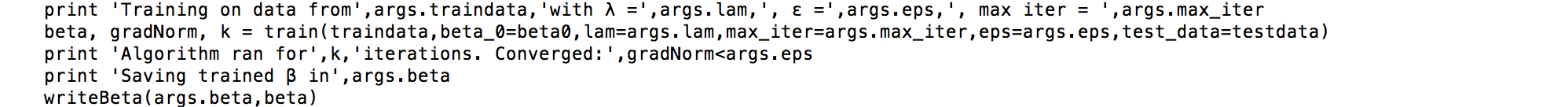
**(a)**

The code I modified is as following:









**(b)**

From the following output, compared with LogisticRegression.py, when executed with λ = 0 over the mushrooms train and test datasets for at most 20 iterations, the results are the same, but the ParallelLogisticRegression.py is faster.

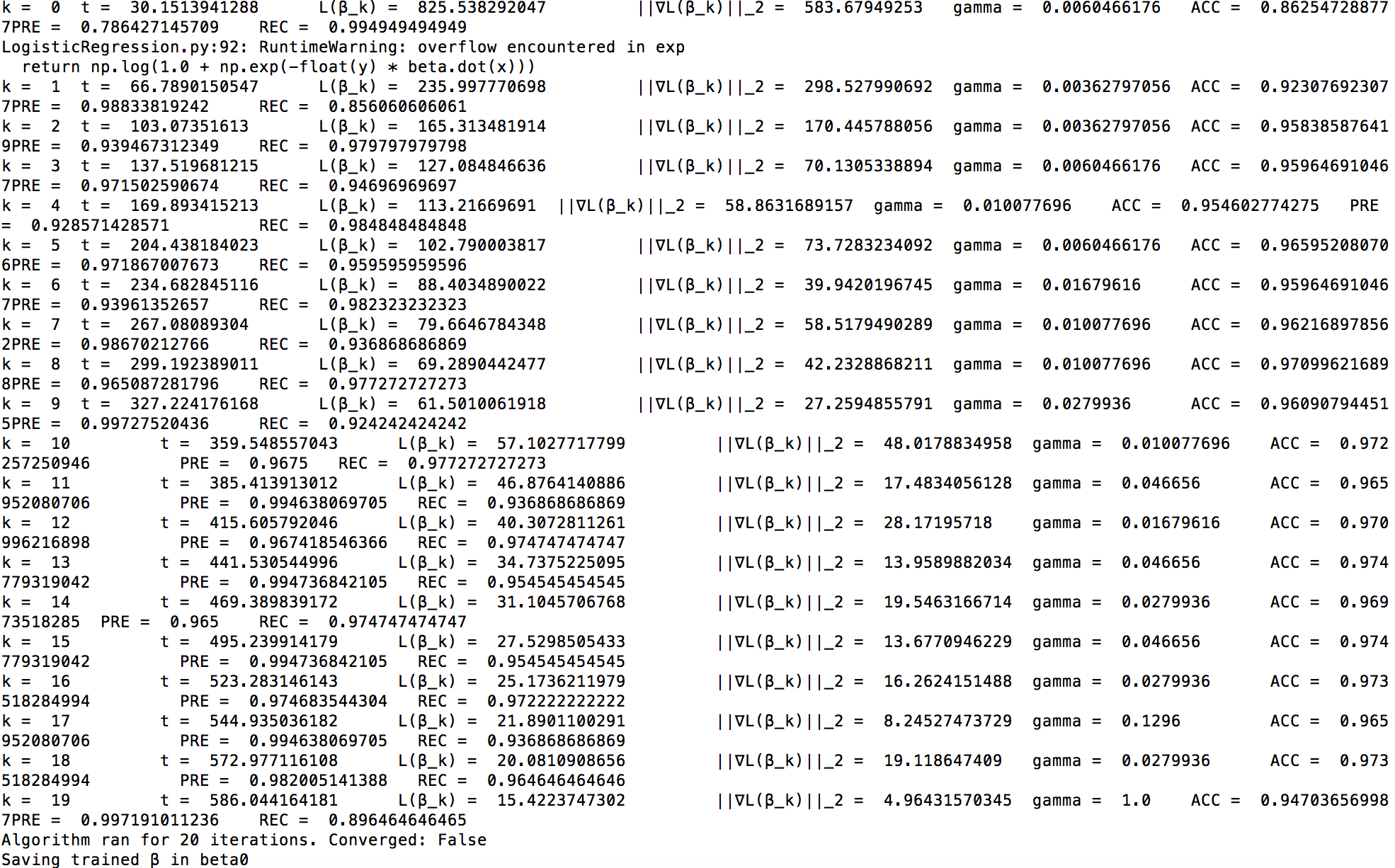
****

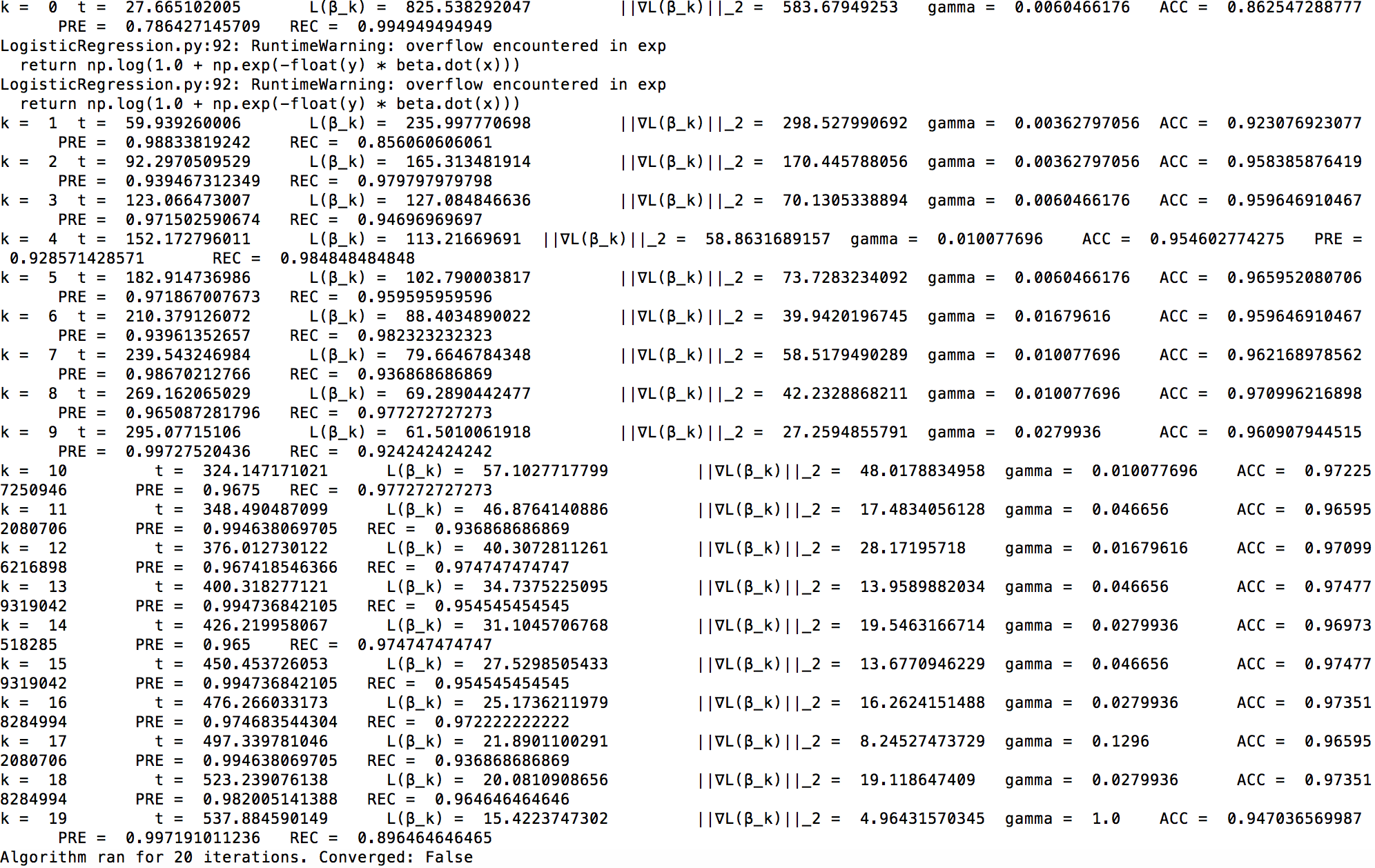
**(c)**

The following is the result we get from execution:

The part in the following screenshot has the effect of “--silent”. We can see that there are two

**(c)**

****Fig 1. logisticRegression

Fig 2. ParallelLogisticRegression

The output of logisticRegression.py and ParallelLogisticRegression.py is shown in Fig 1, and Fig 2, respectively, and from them we can conclude that the outputs are the same, but ParallelLogisticRegression.py is faster.

**Question 5:**

**(a)**

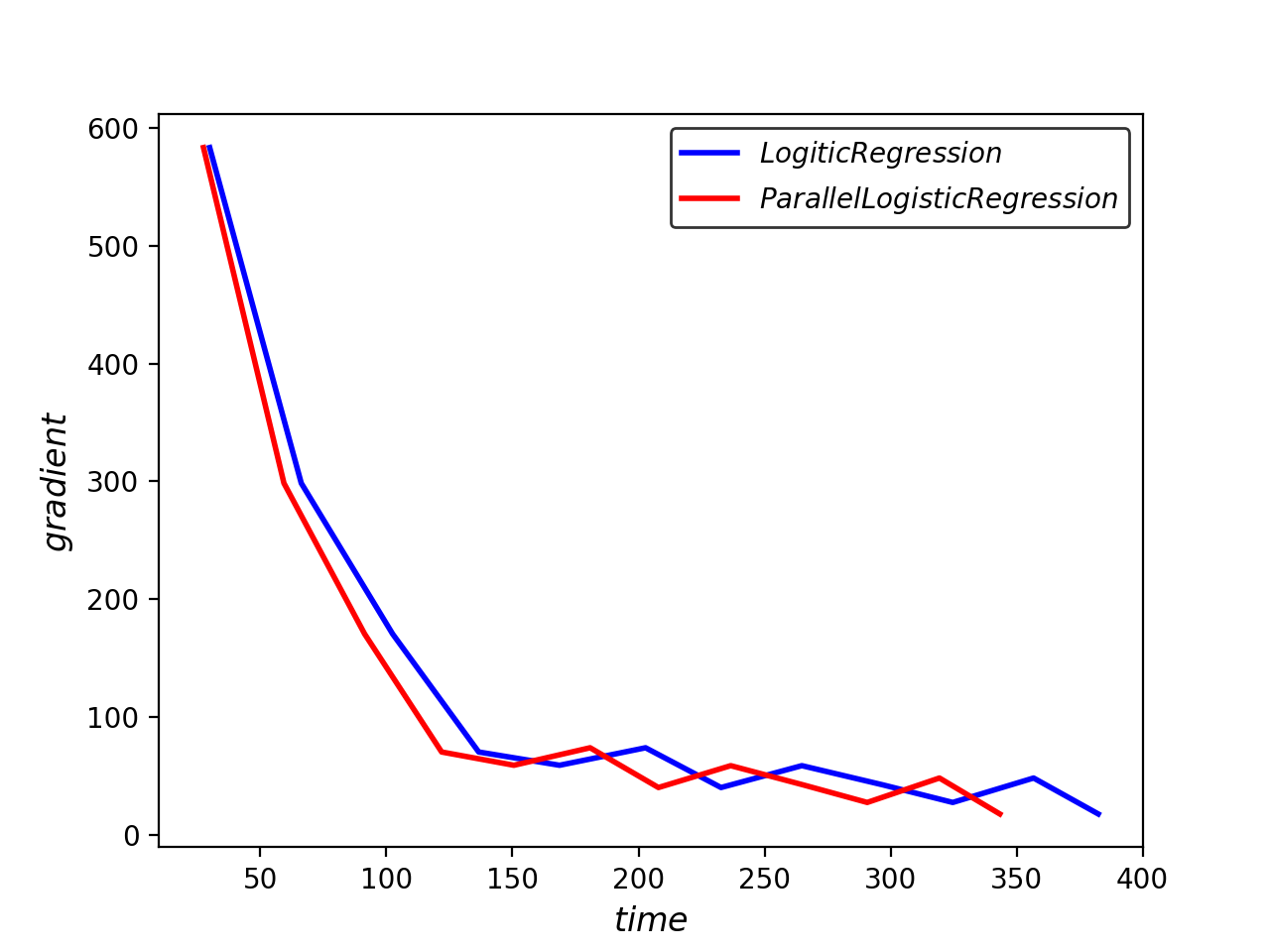


Figure 3 Comparison on gradient

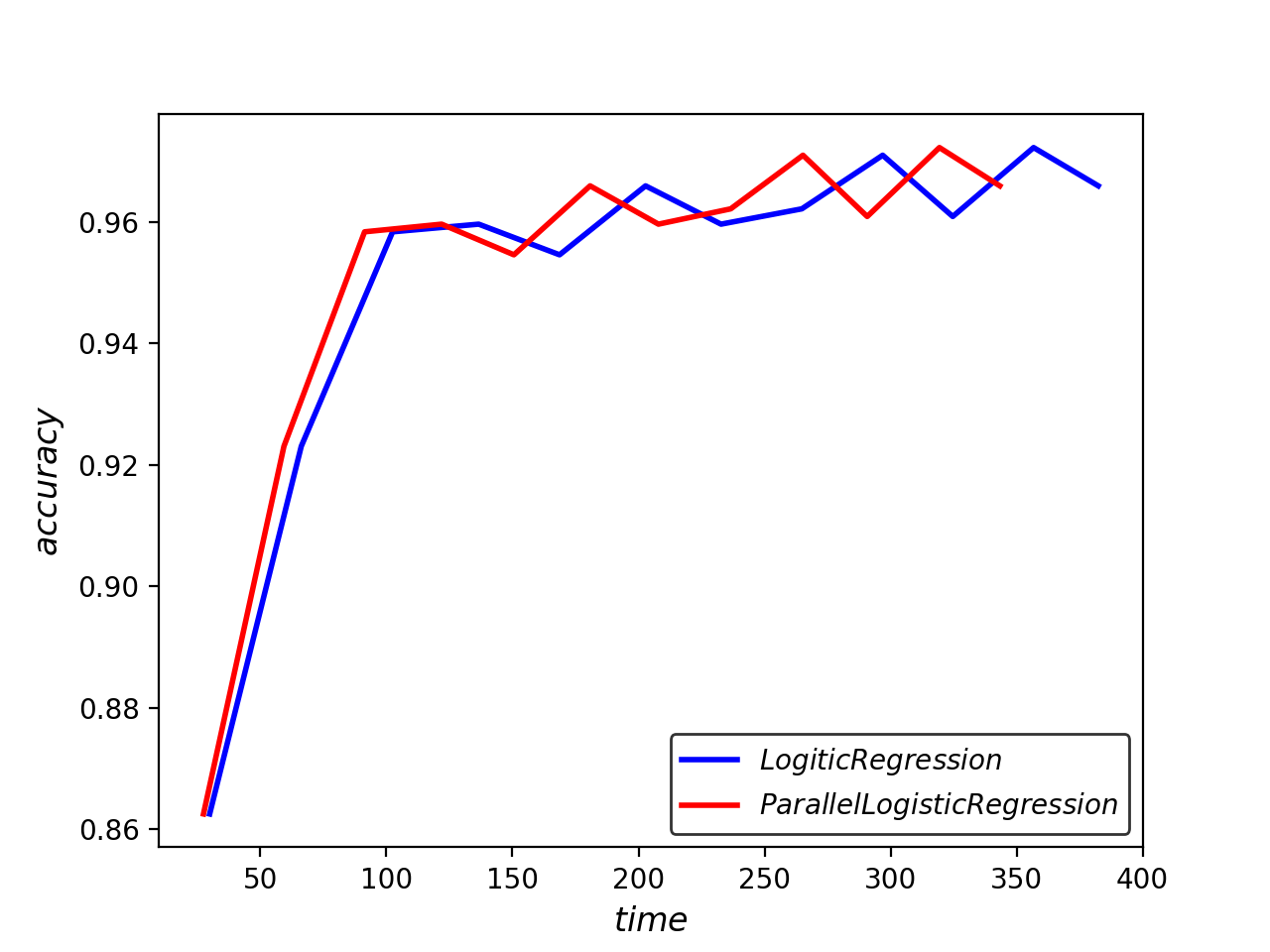


Figure 4 Comparison on accuracy

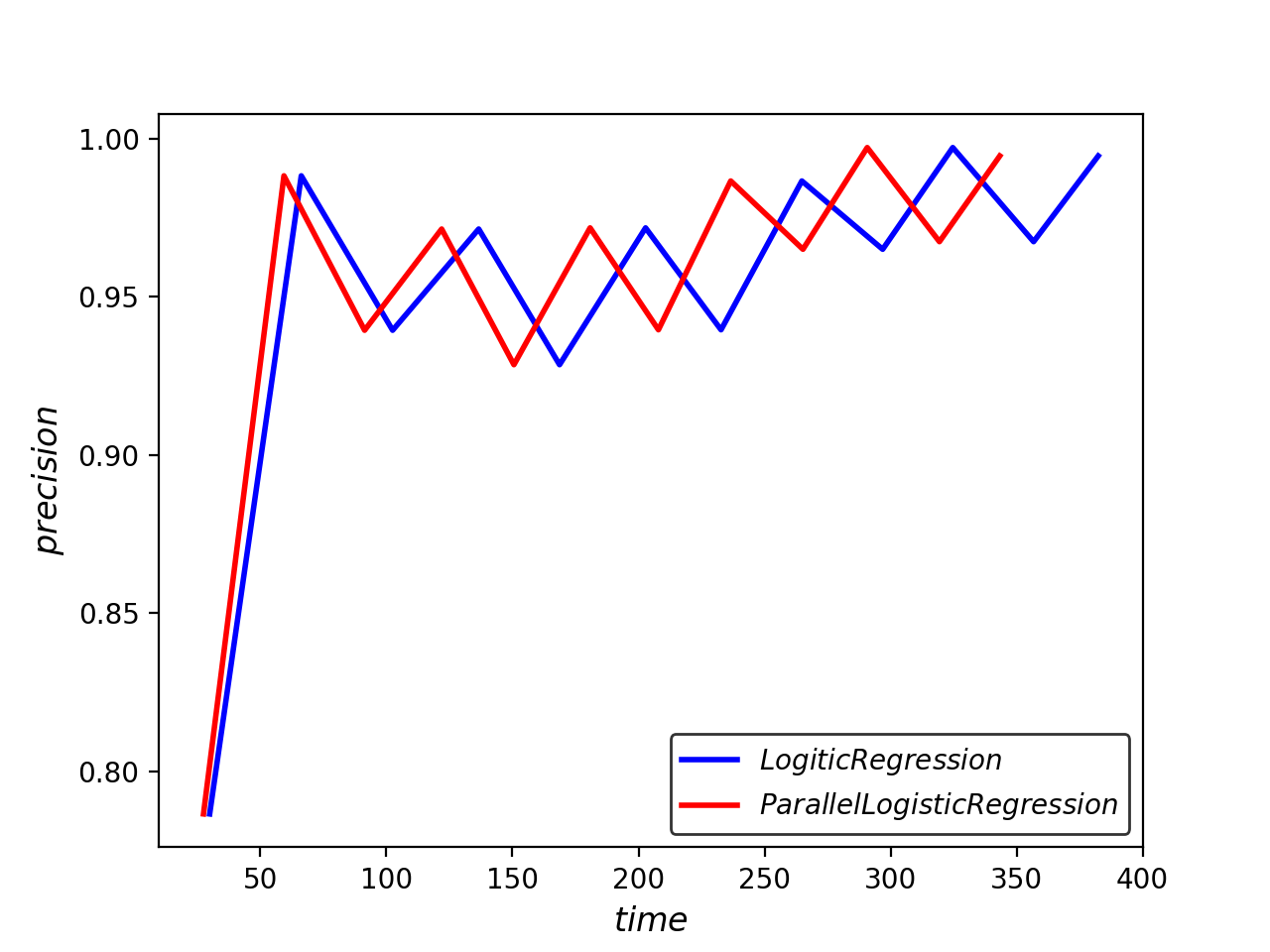


Figure 5 Comparison on precision

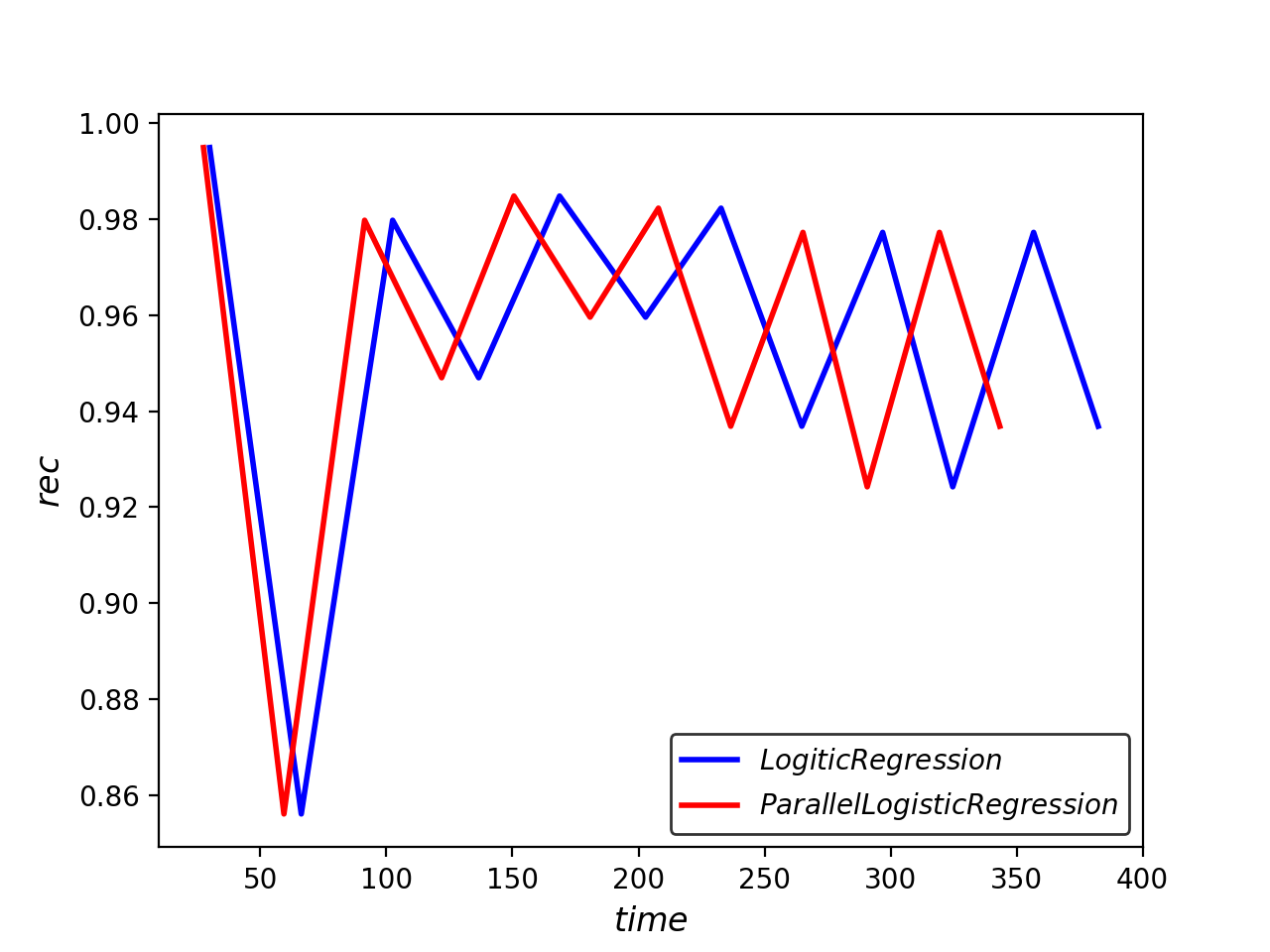


Figure 6 Comparison on recall

**(b)**

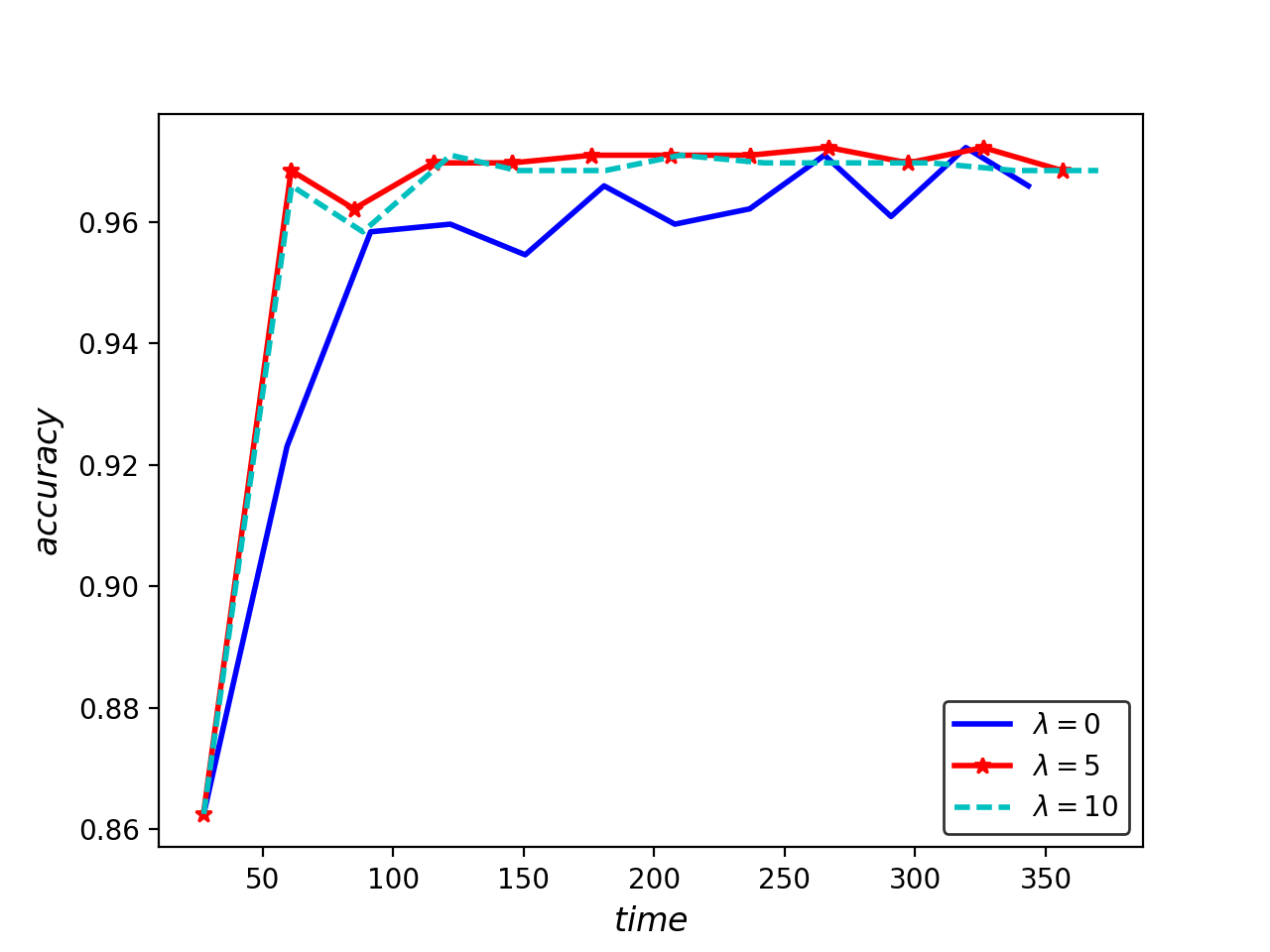


Figure 7 Comparison on accuracy with different lambda

**(c)**

The iteration number I used is just 12, so from the figure in (b), when lambda = 5, the result is better. I believe the result might change if there were more iterations.

For lambda = 5, the features of top 10 positive values are as followings:

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **features** | 'doctor' | 'medic' | 'inform' | 'diseas' | 'treatment' | 'and' | 'effect' | 'gordon' | 'problem' | 'bank' |
| **values** | 0.409115 | 0.365873 | 0.35246 | 0.330965 | 0.308238 | 0.302832 | 0.300839 | 0.289572 | 0.28599 | 0.282045 |

And the features of top 10 negative values are as followings:

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **features** | 'basebal' | 'game' | 'player' | 'team' | 'win' | 'plai' | 'fan' | 'run' | 'pitch' | 'philli' |
| **values** | -0.882569 | -0.725519 | -0.63883 | -0.62689 | -0.456391 | -0.453508 | -0.420471 | -0.41965 | -0.412251 | -0.39157 |