Chapter 5

Regularization

5.1 Learning Objectives

To develop a machine learning algorithm, we typically first select a model, and then learn the model parameters from the training set. For example, in linear regression, we first select the order of features for the model, and then learn parameter vector θ. A complicated relationship between features and labels requires a higher order of features. Similarly, in a logistic regression, a complicated decision boundary requires a higher order of features. However, if the order of features is too high, it will result in *overfitting* – the model fits the training data very well, but cannot generalize to new data examples well.

In this chapter, we will describe the problem of overfitting, which is very common in machine learning, and present a way to deal with it: regularization. After finishing this chapter, one should be able to

- Understand the problem of overfitting, high bias and high variance
- Be aware of options to address overfitting
- Add more features to address underfitting (high bias)
- Understand regularization
- Be able to apply regularization to gradient descent to address overfitting
- Use LogisticRegression() from sklearn for non-regularization or for regularization.
- Be able to plot decision boundary in Python

5.2 Problem of Overfitting

Consider the problem of predicting y from x ∈ R, given a training set. Fig.1(a) shows the result of fitting a linear function $y = \theta_0 + \theta_1 x$ to the training dataset. We see that the data doesn't really lie on straight line, and so the fit is not very good because some data examples are far away from the prediction line. Instead, if we had added an extra feature x^2, and fit $y = \theta_0 + \theta_1 x + \theta_2 x^2$, then we obtain a slightly better fit to the training data, shown in Fig.1(b). However, there is also a problem with adding too many higher order features. For example, Fig.1 (c) shows the result of fitting a 5-order polynomial $y = \theta_0 + \theta_1 x + \theta_2 x^{2+}\theta_3 x^3 + \theta_4 x^4 + \theta_5 x^5$. We see that even though the fitted curve passes through the training data perfectly, we would not expect this to be a very good prediction on a new data example. In other words, the model in Fig. 1(c) does not generalize well. We'll say Fig.1(a) shows an instance of **underfitting or high bias**—in which the data clearly shows structure not captured by the model—and Fig.1(c) is an example of **overfitting or high variance**. The similar phenomena may happen to logistic regression, as shown in Fig.2.

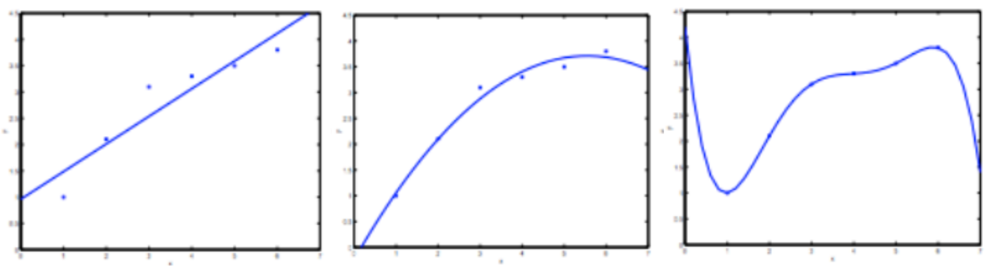


Fig.1 linear regression: a) underfitting, b) good fitting, c) overfitting

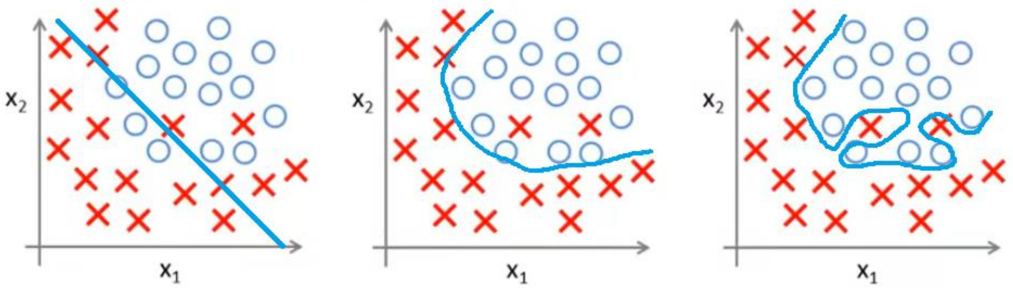


Fig.2 logistic regression: a) underfitting, b) good fitting, c) overfitting

Underfitting, or high bias, happens when the model or hypothesis function h fits poorly to the training data, because the model is too simple. On the other hand, overfitting, or high variance, happens when the model fits the training data very well but does not generalize well to predict new data. It is usually caused by a complicated function that creates a lot of unnecessary curves and angles unrelated to the nature of data. There are two main options to address the issue of overfitting:

1) Reduce the number of features:
   Manually select which features to keep.
   Use a model selection algorithm.
2) Regularization:
   Keep all the features but reduce the magnitude of parameters θ_j. Regularization works well when we have a lot of slightly useful features.

[Example 4.1] **(answer: c)** Consider the medical diagnosis problem of classifying tumors as malignant or benign. If a hypothesis $h_\theta(x)$ has overfit the training set, it means that

a) It makes accurate predictions for examples in the training set and generalizes well to make accurate predictions on new, previously unseen examples.

b) It does not make accurate predictions for examples in the training set, but it does generalize well to make accurate prediction on new, previously unseen examples.

c) It makes accurate predictions for examples in the training set, but it does not generalize well to make accurate prediction on new, previously unseen examples.

d) It does not make accurate predictions for examples in the training set, and does not generalize well to make accurate prediction on new, previously unseen examples.

5.3 Regularization

5.3.1 Cost function with regularization

The reason of an overfitting in a linear regression is that the higher order features contribute too much to the hypothesis function. To alleviate the overfitting problem, we will try to reduce the influence of higher order terms, e.g., $\theta_5 x^5$. An effective way to do this is to reduce the magnitudes of the corresponding parameters θ_j while keeping the cost function (previously defined in gradient descent) close to the minimum. Therefore, when we search for optimal parameters θ without overfitting, we should consider a tradeoff between minimizing the square error and limiting the magnitudes of parameters. For this purpose, the cost function of linear regression can be modified as

$$J(\theta) = \frac{1}{2m}\left[\sum_{i=1}^{m}\left(e^{(i)}\right)^2 + \lambda\sum_{j=1}^{n}\theta_j^2\right]$$

$$= \frac{1}{2m}\left[\sum_{i=1}^{m}\left(h_\theta\left(x^{(i)}\right) - y^{(i)}\right)^2 + \lambda\sum_{j=1}^{n}\theta_j^2\right] = \frac{1}{2m}\left[(\mathbf{X}\cdot\theta - Y)^T(\mathbf{X}\cdot\theta - Y) + \lambda\sum_{j=1}^{n}\theta_j^2\right] \quad (5.1)$$

In the above equation, we regularize all the parameters by adding the penalty of the parameters' magnitudes to the square-error based cost function. To learn the model, we minimize the modified cost function:

$$\min_\theta J(\theta) \quad (5.2)$$

The λ, or lambda, is the regularization parameter. It controls the degree of regularization, i.e., how much costs of the parameters result in. A larger λ will lead to a heavier regularization. Under the regularization, the hypothesis model will have a smooth output, hence with a reduced overfitting. If λ is too larger, it may smooth out the function too much and cause underfitting. On the other hand, if λ is too small or zero, the model is reduced to the case without regularization.

5.3.3 Regularization for linear regression

Now we apply regularization to linear regression through gradient descent. From *Chapter 3: Linear Regression*, we have the gradient of cost function without considering regularization

$$\frac{\partial J(\theta)}{\partial \theta_j} = \frac{1}{m}\sum_{i=1}^{m}\left(h_\theta\left(x^{(i)}\right) - y^{(i)}\right)x_j^{(i)}$$

$$grad = \begin{bmatrix} \frac{\partial J(\theta)}{\partial \theta_0} \\ \frac{\partial J(\theta)}{\partial \theta_1} \\ \vdots \\ \frac{\partial J(\theta)}{\partial \theta_n} \end{bmatrix} = \frac{1}{m} X^T \cdot (X \cdot \theta - Y)$$

Consider the term $\lambda \sum_{j=1}^{n} \theta_j^2$ added to the cost function for regularization (5.1), we can have the gradient with regularization as

$$\frac{\partial J(\theta)}{\partial \theta_0} = \frac{1}{m} \sum_{i=1}^{m} \left(h_\theta\left(x^{(i)}\right) - y^{(i)} \right) x_0^{(i)} \tag{5.3a}$$

$$\frac{\partial J(\theta)}{\partial \theta_j} = \frac{1}{m} \sum_{i=1}^{m} \left(h_\theta\left(x^{(i)}\right) - y^{(i)} \right) x_j^{(i)} + \frac{\lambda}{m} \theta_j \qquad j=1,2,\ \ldots,\ n \tag{5.3b}$$

Its vectorized form is

$$grad = \begin{bmatrix} \frac{\partial J(\theta)}{\partial \theta_0} \\ \frac{\partial J(\theta)}{\partial \theta_1} \\ \vdots \\ \frac{\partial J(\theta)}{\partial \theta_n} \end{bmatrix} = \frac{1}{m} X^T \cdot (X \cdot \theta - Y) + \frac{\lambda}{m} \begin{bmatrix} 0 \\ \theta_1 \\ \vdots \\ \theta_n \end{bmatrix} \tag{5.4}$$

Thus, the gradient descent for linear regression with regularization can be described as a repeating update process of parameters θ as follows:

Repeat {

$$\theta_0 := \theta_0 - \alpha \frac{1}{m} \sum_{i=1}^{m} \left(h_\theta\left(x^{(i)}\right) - y^{(i)} \right) x_0^{(i)} \tag{5.5a}$$

$$\theta_j := \theta_j - \alpha \left[\frac{1}{m} \sum_{i=1}^{m} \left(h_\theta\left(x^{(i)}\right) - y^{(i)} \right) x_j^{(i)} + \frac{\lambda}{m} \theta_j \right] \tag{5.5b}$$

}

With a re-arrange of (5.5b), the repeat process can be represented as

Repeat {

$$\theta_0 := \theta_0 - \alpha \frac{1}{m} \sum_{i=1}^{m} \left(h_\theta\left(x^{(i)}\right) - y^{(i)} \right) x_0^{(i)} \tag{5.6a}$$

$$\theta_j := \left(1 - \frac{\alpha\lambda}{m}\right) \theta_j - \alpha \frac{1}{m} \sum_{i=1}^{m} \left(h_\theta\left(x^{(i)}\right) - y^{(i)} \right) x_j^{(i)} \qquad \text{j=1,2,…n} \tag{5.6b}$$

}

From (5.6a) and (5.6b), we can see that the regularization introduces a factor $\left(1 - \frac{\alpha\lambda}{m}\right) < 1$ on every update, thus reducing the magnitude of θ_j, j=1,2…,n.

5.3.4 Regularization for logistic regression

Like linear regression, we can similarly apply regularization to logistic regression for alleviating overfitting problem in a classification task. The cost function can be modified as

$$J(\boldsymbol{\theta}) = -\frac{1}{m}\sum_{i=1}^{m}[y^{(i)}\ln\left(h_{\boldsymbol{\theta}}(\mathbf{x}^{(i)})\right) + (1-y^{(i)})\ln\left(1-h_{\boldsymbol{\theta}}(\mathbf{x}^{(i)})\right)] + \frac{\lambda}{2m}\sum_{j=1}^{n}\theta_j^2 \qquad (5.7)$$

As a result, we obtain the exact same equations for gradient descent update process as following:

$$\frac{\partial J(\theta)}{\partial \theta_0} = \frac{1}{m}\sum_{i=1}^{m}\left(h_\theta\left(x^{(i)}\right) - y^{(i)}\right)x_0^{(i)} \qquad (5.8a)$$

$$\frac{\partial J(\theta)}{\partial \theta_j} = \frac{1}{m}\sum_{i=1}^{m}\left(h_\theta\left(x^{(i)}\right) - y^{(i)}\right)x_j^{(i)} + \frac{\lambda}{m}\theta_j \qquad j=1,2,\ ....,\ n \qquad (5.8b)$$

Repeat {

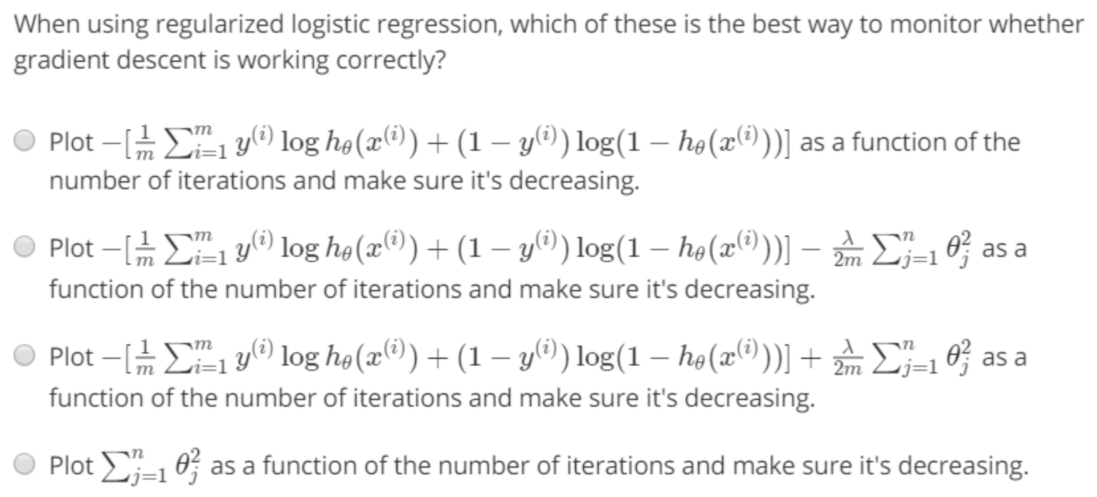
$$\theta_0 := \theta_0 - \alpha\frac{1}{m}\sum_{i=1}^{m}\left(h_\theta\left(x^{(i)}\right) - y^{(i)}\right)x_0^{(i)} \qquad (5.9a)$$

$$\theta_j := \left(1 - \frac{\alpha\lambda}{m}\right)\theta_j - \alpha\frac{1}{m}\sum_{i=1}^{m}\left(h_\theta\left(x^{(i)}\right) - y^{(i)}\right)x_j^{(i)} \qquad \text{j=1,2,…n} \qquad (5.9b)$$

}

Remarks: the hypothesis function $h_\theta\left(x^{(i)}\right)$ in logistic regression is different from the one in linear regression.

[Example 4.2]



When using regularized logistic regression, which of these is the best way to monitor whether gradient descent is working correctly?

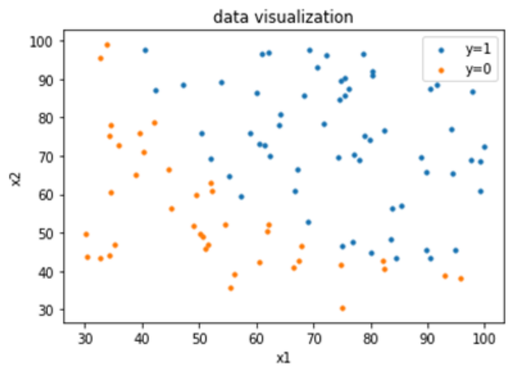
- ○ Plot $-[\frac{1}{m}\sum_{i=1}^{m} y^{(i)}\log h_\theta(x^{(i)}) + (1-y^{(i)})\log(1-h_\theta(x^{(i)}))]$ as a function of the number of iterations and make sure it's decreasing.
- ○ Plot $-[\frac{1}{m}\sum_{i=1}^{m} y^{(i)}\log h_\theta(x^{(i)}) + (1-y^{(i)})\log(1-h_\theta(x^{(i)}))] - \frac{\lambda}{2m}\sum_{j=1}^{n}\theta_j^2$ as a function of the number of iterations and make sure it's decreasing.
- ○ Plot $-[\frac{1}{m}\sum_{i=1}^{m} y^{(i)}\log h_\theta(x^{(i)}) + (1-y^{(i)})\log(1-h_\theta(x^{(i)}))] + \frac{\lambda}{2m}\sum_{j=1}^{n}\theta_j^2$ as a function of the number of iterations and make sure it's decreasing.
- ○ Plot $\sum_{j=1}^{n}\theta_j^2$ as a function of the number of iterations and make sure it's decreasing.

(answer: c)

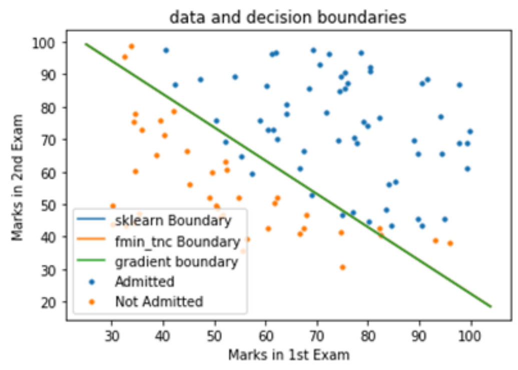
5.4 Practice for Regularization

5.4.1 Fixing underfitting by adding higher-order features (log_reg_add_features.py)

In the previous chapter "Logistic regression", we implemented a classification task that predicts the admission of an applicants based on the scores of two exams (x1, x2). The training set is shown in Fig.3(a). If we only use two features (x1, x2) to train the logistic regression model the decision boundary is linear, shown in Fig.3(b).



(a) Data set



(b) linear decision boundary (accuracy=0.89)

Fig.3. Logistic regression with linear decision boundary

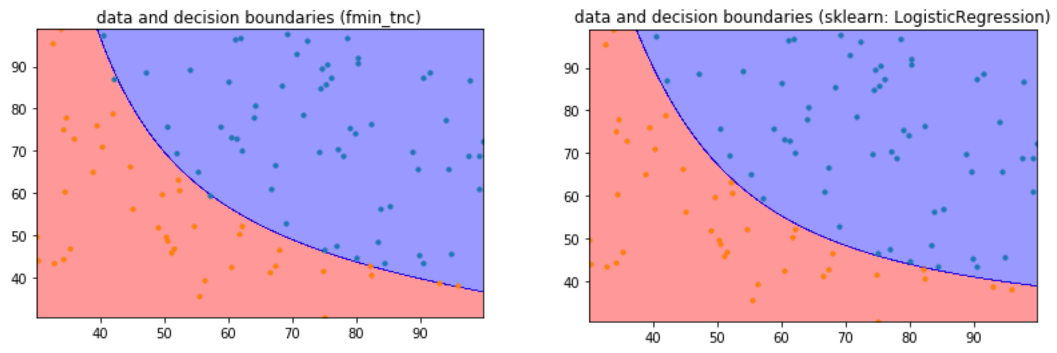
By inspection of the data set, we notice that a non-linear decision boundary may achieve a better accuracy. In this practice, we add more features to training data X so that the resulting decision boundary fits the data better. Specifically, we add columns x_1^2, x_1x_2, x_2^2 to data X, as shown in Fig.4. The hypothesis function is $h_\theta(x) = g(\theta_0 + \theta_1x_1 + \theta_2x_2 + \theta_3x_1^2 + \theta_4x_1x_2 + \theta_5x_2^2)$. First, we use gradient descent with an advanced optimization function fmin_tnc(). The result is shown in Fig.5(a). Second, LogisticRegress() from sklearn library is used for a quick implementation and confirmation. The result is plotted in Fig.5(b). From these results, we can see that adding more feature may fix the problem of underfitting. However, if too many features are added, an overfitting may happen. The next example will demonstrate the overfitting problem and its solution – regularization.

ones	x_1	x_2	y
1	34.6	78.0	0
1	30.2	43.9	0
1	60.2	86.3	1
	…	…	…

Add features

ones	x_1	x_2	x_1^2	x_1x_2	x_2^2	y
1	34.6	78.0	1197.16	….	…	0
1	30.2	43.9	…	…	…	0
1	60.2	86.3	…	…	…	1
…	…	…	…	…	….	…

Fig.4. Add higher-order features to the dataset



(a) quadratic decision boundary
(optimized by fmin_tnc, accuracy=0.95)

(b) quadratic decision boundary
(sklearn: LogisticRegression(), accuracy=1)

Fig.5 A logistic regression classification task with higher-order polynomial features.

The parameter θ (or theta) is generated by the output of log_reg_add_features.py:

```
theta by fmin_tnc: [-1.86643290e-02 -3.75077946e-01 -3.09429917e-01  1.06962099e-03
  1.03079006e-02  3.50023675e-04]
theta
___fmin_tnc: [-1.86643290e-02 -3.75077946e-01 -3.09429917e-01  1.06962099e-03
  1.03079006e-02  3.50023675e-04]
___LogistiRegression: [-0.06202446] [[-8.95942833e-01 -1.40029397e+00 -2.29434515e-04  3.61578679e-02
3.93039309e-03]]

Accuracy
___fmin_tnc: 0.95
___LogisticRegression: 1.0
```

5.4.2 Overfitting due to too many features

Let's consider the dataset (ex2data2.txt) shown in Fig.6(a). It can be seen that dataset cannot be separated into positive and negative examples by a straight-line through the plot. Therefore, a straight-forward application of logistic regression will not perform well on this dataset. One way to fit the data better is to create more features from each data point. We will map the features into all polynomial terms of x_1 and x_2 up to the sixth power.

Ones	x_1	x_2	x_1^2	x_1x_2	x_2^2	x_1^3	…	$x_1x_2^5$	x_2^6	y
1										0
1	Original dataset						28-dimensional features			0
1										1
…										…

As a result of this mapping, our vector of two features has been transformed into a 28-dimensional vector, as shown in the above table. A logistic regression using gradient descent trained on this higher-dimension feature vector will have a more complex decision boundary and will appear nonlinear. While the feature mapping allows us to build a more expressive classifier, it is also more susceptible to overfitting.

We implement the logistic regression in two ways: 1) gradient descent with fmin_tnc optimization (from the scratch); 2) using LogisticRegression () from sklearn.

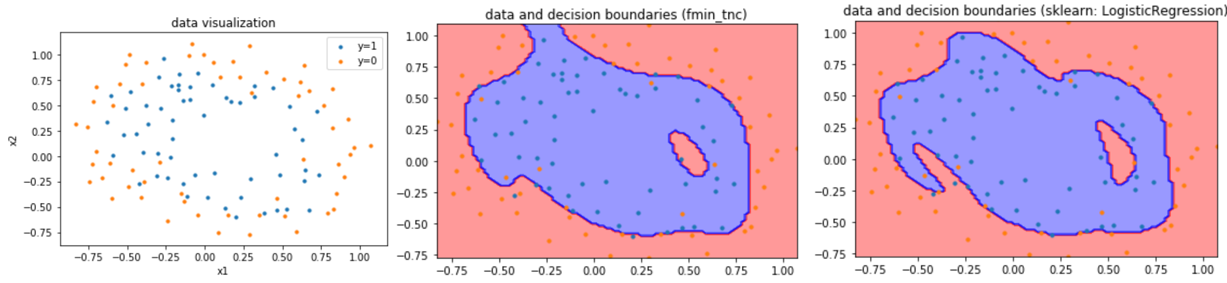


Fig.6 Overfitting due to too many features, (a) original dataset, (b) gradient descent trained on 28 polynomial (up to 6th power) data features, (c) LogisticRegression() on 28 polynomial (up to 6th power) data features.

Remarks: LogistiRegression() has an option for regularization setting. For comparison with gradient descent without regularization, we set LogisticRegression() with no regularization at this moment as:

```
model = LogisticRegression(penalty='none', solver='newton-cg')
```

The decision boundaries are visualized in Fig.6 (b) and (c). Both cases show the overfitting problem because they don't generalize well on new data. The resulting parameter theta and accuracy are printed as

```
theta
___fmin_tnc: [   15.17402669    19.85026422     7.78255634  -163.00167352
  -96.17625296   -59.73117283  -159.31560742  -153.15089914
  -82.95561243     5.75921974   579.97395717   587.77048912
  683.7220233    315.21310069    68.23870008   286.78948423
  453.56742802   525.34101678   312.18150944   108.07725706
  -14.87042815  -669.78634014  -960.59485951 -1384.95203136
 -1118.02828779  -895.97571763  -351.91242647   -31.16305991]
___LogistiRegression: [38.22804169] [[   55.59208832    98.13839434  -369.40296579  -177.1054916
  -194.24615123  -365.98431043  -842.14061998  -719.38840995
  -511.8516527   1182.61829248  1279.23974085  1907.75454796
   914.24226885   514.24059059   573.18578543  1629.6217522
  2553.40262799  2918.82742391  1780.46086969   785.25153167
 -1257.82952594 -2259.83118287 -4142.47380965 -4290.24224096
 -4229.32516445 -2055.38360482  -750.32334206]]

Accuracy
___fmin_tnc: 0.8728813559322034
___LogisticRegression: 0.8898305084745762
```

5.4.3 Regularization to reducing overfitting

If we want to alleviate the overfitting problem while keeping all 28 features, regularization is a solution. We modify cost function and gradient by penalizing parameter θ, according to equation (5.7) and (5.8).

```
def cost_function_regularization(theta, x, y, lamda):
    # Computes the cost function for all the training samples
    m = x.shape[0]
    t1=np.dot(np.transpose((np.log(h(theta,X)))),y)
    t2=np.dot(np.transpose((np.log(1-h(theta,X)))),1-y)
    total_cost = -(1 / m) * (t1+t2)+(0.5*lamda/m)*(np.dot(theta, theta)-theta[0]*theta[0])
    return total_cost
def gradient_regularization(theta, x, y, lamda):
    # Computes the gradient of the cost function at the point theta
    m = x.shape[0]
    temp = (1 / m) * np.dot(x.T, h(theta,x) - y)+(lamda/m)*theta
    temp[0]=temp[0]-(lamda/m)*theta[0]
    return temp
```

Then, gradient descent with fmin_tnc optimization is applied to train a logistic regression classifier on the 28-feature dataset. The decision boundary ($\lambda = 1$) is plotted in Fig.7(a). To confirm that correctness of the result, we also implement a logistic regression classifier with regularization using LogisticRegress () by

```
model = LogisticRegression(penalty='l2', solver='newton-cg')
```

The resulting decision boundary is plotted in Fig.7(b), which is almost the same as Fig.7(a).

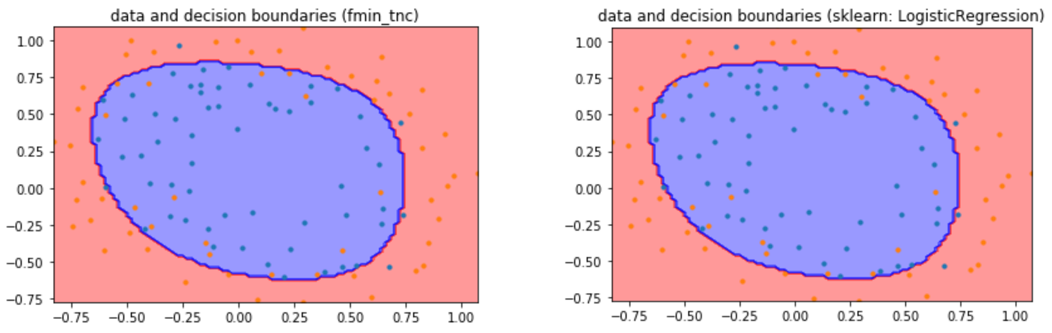


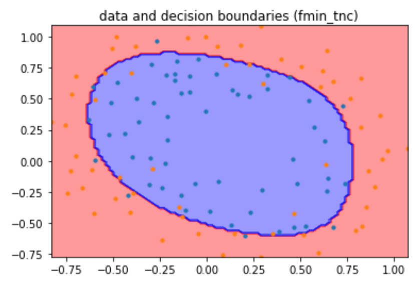
Fig.7 Regularization to reduce overfitting: (a) gradient descent ($\lambda = 1$), (b) LogisticRegression()

The learned parameters and model accuracies in two methods are printed as

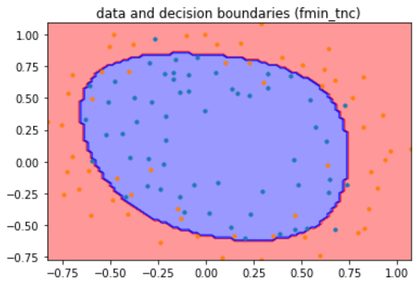
```
theta
___fmin_tnc: [ 1.27271027  0.62529965  1.18111686 -2.01987399 -0.91743189 -1.43166929

  0.12393227 -0.36553118 -0.35725403 -0.17516291 -1.4581701  -0.05098418
 -0.61558551 -0.27469165 -1.19271297 -0.2421784  -0.20603295 -0.04466179
 -0.27778953 -0.29539513 -0.45645982 -1.04319155  0.02779373 -0.29244874
  0.01555761 -0.32742407 -0.14389151 -0.92467487]
___LogistiRegression: [1.27273852] [[ 0.62527427  1.18107953 -2.01995701 -0.91743361 -1.43166228
0.12400943
  -0.36552879 -0.35723375 -0.1751281  -1.45816817 -0.05099315 -0.61556795
  -0.27470949 -1.19281161 -0.24218951 -0.20599958 -0.04473522 -0.27778736
  -0.29537501 -0.45635027 -1.04321271  0.02777197 -0.29243756  0.0155633
  -0.32738395 -0.14388956 -0.92464266]]

Accuracy
___fmin_tnc: 0.8305084745762712
___LogisticRegression: 0.8305084745762712
```

Compared to the results without regularization in Section 5.4.2, the magnitudes of theta are much smaller. The reason is that the penalty of theta magnitudes is added to cost function. Furthermore, the accuracy on the training set is around 0.83, which is lower than the accuracy (0.87 or 0.89) of non-regularization. But regularization leads to a better generalization on new data examples.

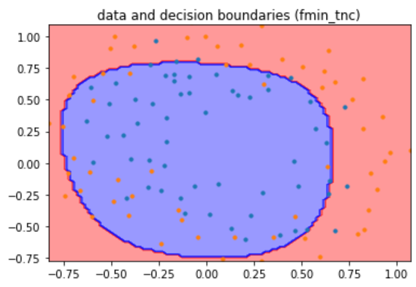
Now, let's explore the effect of λ. We choose: 1)$\lambda = 0.1$ (accuracy=0.839) ; 2)$\lambda = 1$ (accuracy=0.831); 3) $\lambda = 10$ (accuracy=0.745), 4)$\lambda = 100$ (accuracy=0.610)



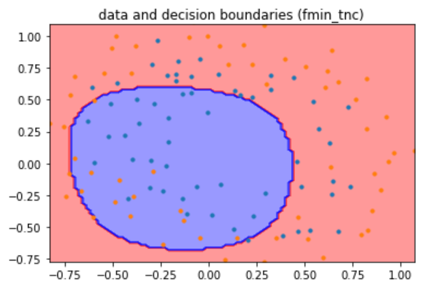
(a) $\lambda = 0.1$, accuracy = 0.839



(b) $\lambda = 1$, accuracy = 0.831



(c) $\lambda = 10$, accuracy = 0.745



(d) $\lambda = 100$, accuracy = 0.610

Fig.8 The effect of λ. $\lambda = 0$: no regularization (Fig.6(b),overfitting), λ too large (e.g.>10): underfitting

The python codes for Section 5.4.2 and Section 5.4.3 are included in log_reg_add_features_regularization.py attached in Appendix.

Summary

In this chapter, we learned the concepts of overfitting (high variance) and underfitting (high bias) and explained the reason for these problems. Underfitting happens when a learning model is too simple to capture the characteristics of the dataset. For instance, if we want to fit the dataset approximately following quadratic shape using a linear model, a high bias (or underfitting) will occur. On the contrast, if the learning model is too complicated (e.g. too many parameters) for the dataset, the model will not generalize well on new data examples though it achieves a high accuracy on the training set. An overfitting model captures too much noise details of the training set.

Then, a mathematical description of regularization for both linear regression and logistic regression is presented. The penalty of parameter magnitude is added to the cost function to reduce the parameter magnitude while keeping the prediction error almost minimum.

Finally, in practice, we use a few examples through Python to demonstrate the concepts of underfitting, overfitting, and regularization. Adding more features can address the underfitting issue.

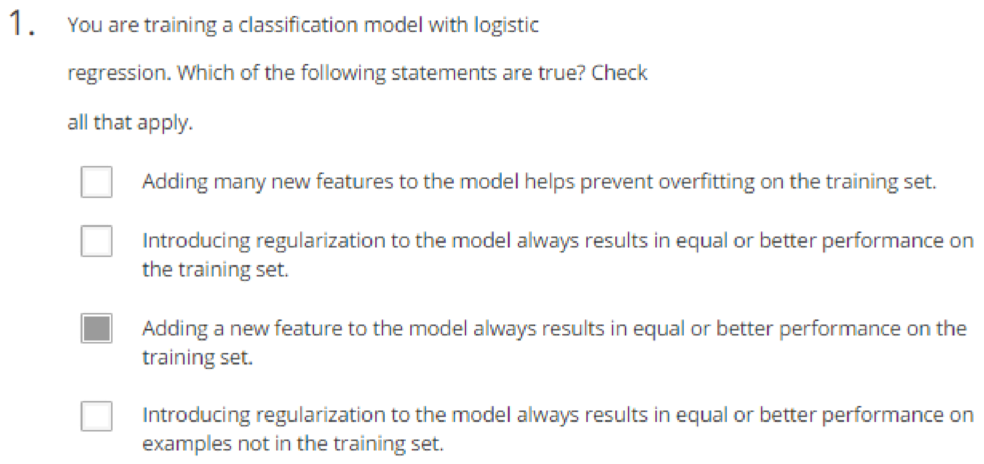
However, adding too many features may lead to overfitting. To address the overfitting problem, regularization can be applied. All examples have been implemented in two ways: 1) gradient descent with fmin_tnc optimization and 2) sklearn LogisticRegression(). For LogisticRegression(), we can specify whether a regularization is applied or not, but cannot specify the value of λ.

1) Example without regularization

   model = LogisticRegression(penalty='none', solver='newton-cg')

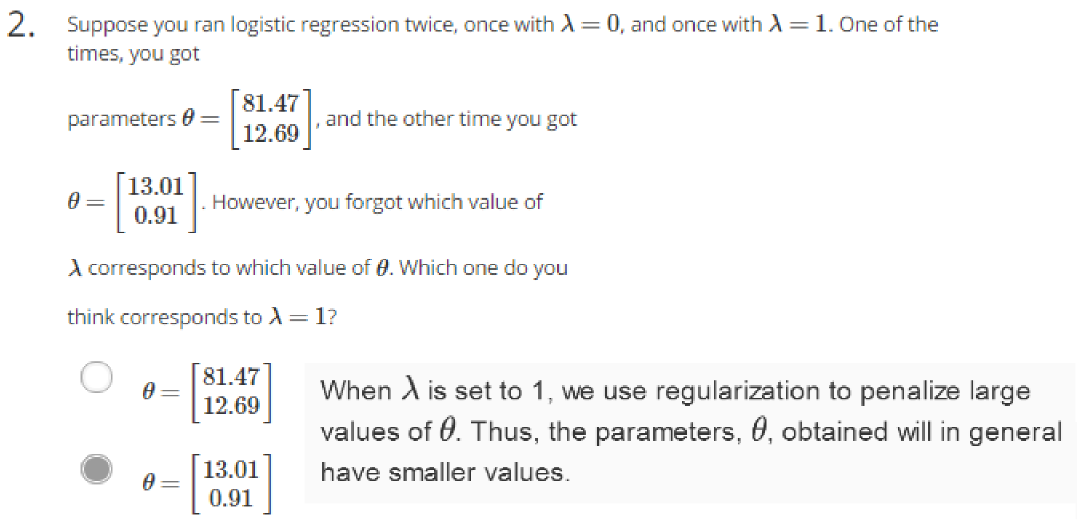
2) Example with regularization
   model = LogisticRegression(penalty='l2', solver='newton-cg')

Exercises



1. You are training a classification model with logistic regression. Which of the following statements are true? Check all that apply.

   - ☐ Adding many new features to the model helps prevent overfitting on the training set.
   - ☐ Introducing regularization to the model always results in equal or better performance on the training set.
   - [✓ Adding a new feature to the model always results in equal or better performance on the training set.]
   - ☐ Introducing regularization to the model always results in equal or better performance on examples not in the training set.



2. Suppose you ran logistic regression twice, once with $\lambda = 0$, and once with $\lambda = 1$. One of the times, you got

   parameters $\theta = \begin{bmatrix} 81.47 \\ 12.69 \end{bmatrix}$, and the other time you got

   $\theta = \begin{bmatrix} 13.01 \\ 0.91 \end{bmatrix}$. However, you forgot which value of

   λ corresponds to which value of θ. Which one do you

   think corresponds to $\lambda = 1$?

   - ○ $\theta = \begin{bmatrix} 81.47 \\ 12.69 \end{bmatrix}$
   - [✓ $\theta = \begin{bmatrix} 13.01 \\ 0.91 \end{bmatrix}$]

   When λ is set to 1, we use regularization to penalize large values of θ. Thus, the parameters, θ, obtained will in general have smaller values.

3. Which of the following statements about regularization are

true? Check all that apply.

[✓] Using too large a value of λ can cause your hypothesis to underfit the data.

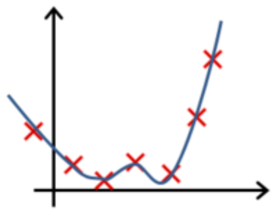
[ ] Because regularization causes $J(\theta)$ to no longer be convex, gradient descent may not always converge to the global minimum (when $\lambda > 0$, and when using an appropriate learning rate α).

[ ] Because logistic regression outputs values $0 \leq h_\theta(x) \leq 1$, it's range of output values can only be "shrunk" slightly by regularization anyway, so regularization is generally not helpful for it.

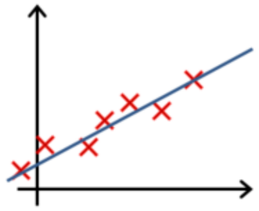
[ ] Using a very large value of λ cannot hurt the performance of your hypothesis; the only reason we do not set λ to be too large is to avoid numerical problems.

4. In which one of the following figures do you think the hypothesis has overfit the training set?

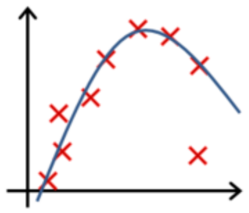
[✓] Figure:



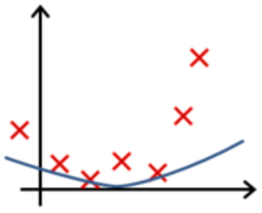
[ ] Figure:



[ ] Figure:



[ ] Figure:



5. In which one of the following figures do you think the hypothesis has underfit the training set?

[✓ Figure:]

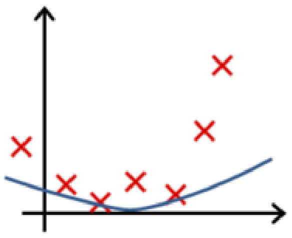


Figure:

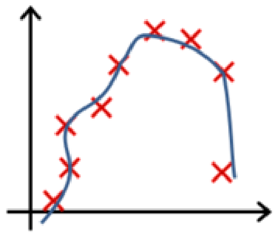


Figure:

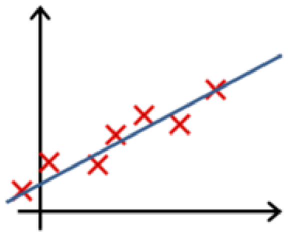
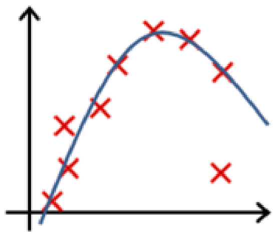


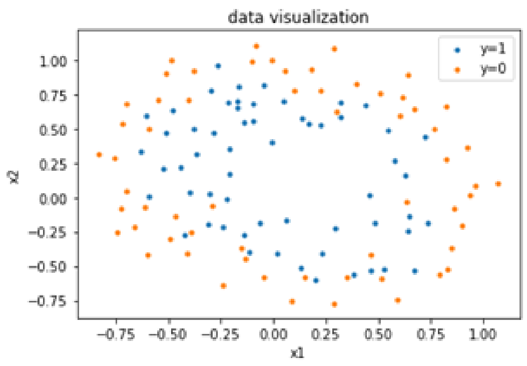
Figure:



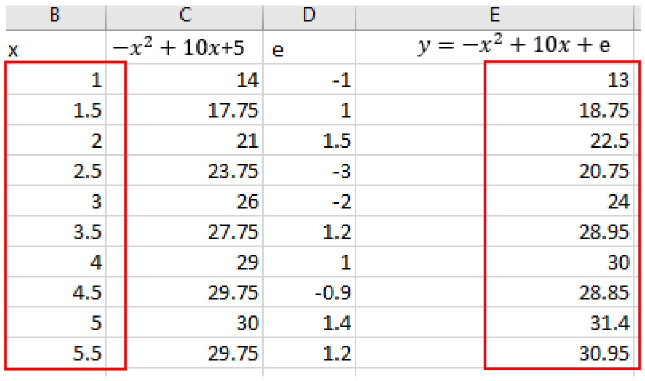
6. (programming) The training dataset ex2data2.txt is plotted in the figure below. Learn a regression model

$$h_\theta(x) = g(\theta_0 + \theta_1 x_1 + \theta_2 x_2 + \theta_3 x_1^2 + \theta_4 x_1 x_2 + \theta_5 x_2^2).$$

Plot the decision boundary on the data scatterplot, and find the accuracy. (hint: modify log_reg_add_features.py)



7. (programming) Suppose 10 training date examples (x,y) are generated by Excel, highlighted by red frame below.



B	C	D	E
x	$-x^2 + 10x$+5	e	$y = -x^2 + 10x + \text{e}$
1	14	-1	13
1.5	17.75	1	18.75
2	21	1.5	22.5
2.5	23.75	-3	20.75
3	26	-2	24
3.5	27.75	1.2	28.95
4	29	1	30
4.5	29.75	-0.9	28.85
5	30	1.4	31.4
5.5	29.75	1.2	30.95

1) Learn a linear regression model $h_\theta(x) = \theta_0 + \theta_1 x$ to fit the 10 data examples.

2) Learn a linear regression model $h_\theta(x) = \theta_0 + \theta_1 x + \theta_2 x^2$ to fit the 10 data examples.

3) Learn a linear regression model $h_\theta(x) = \theta_0 + \theta_1 x + \theta_2 x^2 + \theta_3 x^3 + \theta_4 x^4$ to fit the 10 data examples.

4) Determine whether the model in 3) is overfitting. If yes, train a regularized model for $h_\theta(x) = \theta_0 + \theta_1 x + \theta_2 x^2 + \theta_3 x^3 + \theta_4 x^4$.

Appendix

1) **log_reg_add_features.py**

```
# # -*- coding: utf-8 -*-
"""
Created on Fri Apr  5 09:22:00 2019
log_reg_add_features.py
@author: ml1444
"""

import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
from matplotlib.colors import ListedColormap

# load data by pd.read_csv
dataset_pd = pd.read_csv('marks.txt', header=None)

# visualize the data
y_pd=dataset_pd.iloc[:,-1]
X_pd=dataset_pd.iloc[:,:-1]
X_admit = X_pd.loc[y_pd==1]
X_not_admit = X_pd.loc[y_pd==0]
plt.scatter(X_admit.iloc[:, 0], X_admit.iloc[:, 1], s=10, label='y=1')
plt.scatter(X_not_admit.iloc[:, 0], X_not_admit.iloc[:, 1], s=10, label='y=0')
plt.legend()
plt.xlabel('x1')
plt.ylabel('x2')
plt.title('data visualization')
plt.show()

# prepare data in np.ndarray format

X_temp=X_pd.to_numpy()

# adding more features x1**2, x1*x2, x2**2
x1_square=np.multiply(X_temp[:,0], X_temp[:,0])
x1_x2=np.multiply(X_temp[:,0], X_temp[:,1])
x2_square=np.multiply(X_temp[:,1], X_temp[:,1])
X=np.c_[np.ones((X_pd.shape[0],1)), X_temp, x1_square, x1_x2, x2_square]
#  X: np.ndarray (100,6)  (m,n+1)

y = y_pd[:, np.newaxis]                      # y: np.ndarray (100,1)  (m,1)
theta = np.zeros((X.shape[1], 1))           # theta: np.ndarray (6,1) (n+1,1)

def sigmoid(x):
   return 1 / (1 + np.exp(-x))

def net_input(theta, x):
    # Computes the weighted sum of inputs
    return np.dot(x, theta)
def h(theta, x):
    # Returns the probability after passing through sigmoid
    return sigmoid(net_input(theta, x))

def cost_function(theta, x, y):
    # Computes the cost function for all the training samples
    m = x.shape[0]
    t1=np.dot(np.transpose((np.log(h(theta,X)))),y)
    t2=np.dot(np.transpose((np.log(1-h(theta,X)))),1-y)
```

```python
        total_cost = -(1 / m) * (t1+t2)
        return total_cost

    def gradient(theta, x, y):
        # Computes the gradient of the cost function at the point theta
        m = x.shape[0]
        return (1 / m) * np.dot(x.T, h(theta,x) - y)



    # advanced optimization by scipy fmin_tnc
    from scipy.optimize import fmin_tnc
    def fit(self, x, y, theta):
        opt_weights = fmin_tnc(func=cost_function, x0=theta,
                      fprime=gradient,args=(x, y.flatten()))
        return opt_weights[0]

    parameters = fit(fit, X, y, theta)
    print("theta by fmin_tnc:", parameters)


    # predict and accuracy

    h_pred=np.dot(X,np.reshape(parameters,(6,1)))
    y_pred=[h_pred>0]
    correct=[y_pred==y]
    accuracy_fmin_tnc=np.sum(correct)/len(correct[0][0])

    # sklearn LogisticRegression
    from sklearn.linear_model import LogisticRegression
    from sklearn.metrics import accuracy_score
    model = LogisticRegression(solver='lbfgs')
    model.fit(X[:,1:], np.ndarray.flatten(y))  # data array without ONE column
    predicted_classes = model.predict(X[:,1:])
    accuracy_sklearn = accuracy_score(y.flatten(),predicted_classes)
    par_sklearn = model.coef_
    intercept= model.intercept_

    # Print the summary of resutls
    print('theta')
    print('___fmin_tnc:', parameters)
    print('___LogistiRegression:', intercept, par_sklearn, '\n')
    print('Accuracy')
    print('___fmin_tnc:', accuracy_fmin_tnc)
    print('___LogisticRegression:',accuracy_sklearn)


    # plot decision boundary
    colors = ('red','blue','lightgreen','gray','cyan')
    cmap = ListedColormap(colors[:len(np.unique(y))])

    x1_min, x1_max = X[:,1].min()-0,X[:,1].max()+0
    x2_min, x2_max = X[:,2].min()-0,X[:,2].max()+0
    resolution=0.02
    xx1, xx2 = np.meshgrid(np.arange(x1_min, x1_max, resolution), np.arange(x2_min, x2_max, re
    solution))

    temp = np.array([xx1.ravel(),xx2.ravel()]).T

    x1_square=np.multiply(temp[:,0], temp[:,0])
    x1_x2=np.multiply(temp[:,0], temp[:,1])
```

```
x2_square=np.multiply(temp[:,1], temp[:,1])
XX=np.c_[np.ones((temp.shape[0],1)), temp, x1_square, x1_x2, x2_square]

# plot decision boundary for fmin_tnc method
h_pred=np.dot(XX,np.reshape(parameters,(6,1)))
h_pred=h_pred.reshape(xx1.shape)
y_pred=np.array([h_pred>0]).reshape(xx1.shape)

plt.contourf(xx1,xx2,y_pred, alpha=0.4, cmap=cmap)
plt.xlim(xx1.min(),xx1.max())
plt.ylim(xx2.min(),xx2.max())
plt.scatter(X_admit.iloc[:, 0], X_admit.iloc[:, 1], s=10, label='Admitted')
plt.scatter(X_not_admit.iloc[:, 0], X_not_admit.iloc[:, 1], s=10, label='Not Admitted')
plt.title('data and decision boundaries (fmin_tnc)')
plt.show()

# plot decision boundary for sklearn LogisticRegression() method
parameters_sklearn=np.c_[intercept, par_sklearn].flatten()
h_pred=np.dot(XX,np.reshape(parameters_sklearn,(6,1)))
h_pred=h_pred.reshape(xx1.shape)
y_pred=np.array([h_pred>0]).reshape(xx1.shape)

plt.contourf(xx1,xx2,y_pred, alpha=0.4, cmap=cmap)
plt.xlim(xx1.min(),xx1.max())
plt.ylim(xx2.min(),xx2.max())
plt.scatter(X_admit.iloc[:, 0], X_admit.iloc[:, 1], s=10, label='Admitted')
plt.scatter(X_not_admit.iloc[:, 0], X_not_admit.iloc[:, 1], s=10, label='Not Admitted')
plt.title('data and decision boundaries (sklearn: LogisticRegression)')
plt.show()
```

2) log_reg_add_features_regularization.py

```
# -*- coding: utf-8 -*-
"""
Created on Fri Apr  5 09:22:00 2019
log_reg_add_features_regularization.py
@author: ml1444
"""

import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
from matplotlib.colors import ListedColormap


def sigmoid(x):
   return 1 / (1 + np.exp(-x))

def net_input(theta, x):
    # Computes the weighted sum of inputs
    return np.dot(x, theta)
def h(theta, x):
    # Returns the probability after passing through sigmoid
    return sigmoid(net_input(theta, x))

def cost_function(theta, x, y):
    # Computes the cost function for all the training samples
    m = x.shape[0]
```

```
        t1=np.dot(np.transpose((np.log(h(theta,X)))),y)
        t2=np.dot(np.transpose((np.log(1-h(theta,X)))),1-y)
        total_cost = -(1 / m) * (t1+t2)
        return total_cost

    def gradient(theta, x, y):
        # Computes the gradient of the cost function at the point theta
        m = x.shape[0]
        return (1 / m) * np.dot(x.T, h(theta,x) - y)

    def cost_function_regularization(theta, x, y, lamda):
        # Computes the cost function for all the training samples
        m = x.shape[0]
        t1=np.dot(np.transpose((np.log(h(theta,X)))),y)
        t2=np.dot(np.transpose((np.log(1-h(theta,X)))),1-y)
        total_cost = -(1 / m) * (t1+t2)+(0.5*lamda/m)*(np.dot(theta, theta)-theta[0]*theta[0])
        return total_cost
    def gradient_regularization(theta, x, y, lamda):
        # Computes the gradient of the cost function at the point theta
        m = x.shape[0]
        temp = (1 / m) * np.dot(x.T, h(theta,x) - y)+(lamda/m)*theta
        temp[0]=temp[0]-(lamda/m)*theta[0]
        return temp
    def feature_map(X_temp):
        # add up to 6th power polynomial features
        x1=X_temp[:,0]
        x2=X_temp[:,1]
        # adding more features x1**2, x1*x2, x2**2 ...
        x1_2=np.power(x1,2)
        x1_x2=np.multiply(x1, x2)
        x2_2=np.power(x2,2)

        x1_3=np.power(x1,3)
        x1_2_x2=np.multiply(x1_2, x2)
        x1_x2_2=np.multiply(x1, x2_2)
        x2_3=np.power(x2,3)

        x1_4=np.power(x1,4)
        x1_3_x2=np.multiply(x1_3, x2)
        x1_2_x2_2=np.multiply(x1_2,x2_2)
        x1_x2_3=np.multiply(x1,x2_3)
        x2_4=np.power(x2,4)

        x1_5=np.power(x1,5)
        x1_4_x2=np.multiply(x1_4,x2)
        x1_3_x2_2=np.multiply(x1_3,x2_2)
        x1_2_x2_3=np.multiply(x1_2, x2_3)
        x1_x2_4=np.multiply(x1, x2_4)
        x2_5=np.power(x2,5)

        x1_6=np.power(x1,6)
        x1_5_x2=np.multiply(x1_5, x2)
        x1_4_x2_2=np.multiply(x1_4, x2_2)
        x1_3_x2_3=np.multiply(x1_3, x2_3)
        x1_2_x2_4=np.multiply(x1_2, x2_4)
        x1_x2_5=np.multiply(x1, x2_5)
        x2_6=np.power(x2, 6)


```

```python
    X=np.c_[np.ones((X_temp.shape[0],1)), X_temp, x1_2, x1_x2, x2_2, x1_3, x1_2_x2,x1_x2_2
,
            x2_3, x1_4, x1_3_x2, x1_2_x2_2, x1_x2_3, x2_4,
            x1_5, x1_4_x2, x1_3_x2_2, x1_2_x2_3, x1_x2_4, x2_5,
            x1_6, x1_5_x2, x1_4_x2_2, x1_3_x2_3,x1_2_x2_4, x1_x2_5, x2_6]
    return X
    # X: (*, 28)

# load data by pd.read_csv
dataset_pd = pd.read_csv('ex2data2.txt', header=None)

# visualize the data
y_pd=dataset_pd.iloc[:,-1]
X_pd=dataset_pd.iloc[:,:-1]
X_admit = X_pd.loc[y_pd==1]
X_not_admit = X_pd.loc[y_pd==0]
plt.scatter(X_admit.iloc[:, 0], X_admit.iloc[:, 1], s=10, label='y=1')
plt.scatter(X_not_admit.iloc[:, 0], X_not_admit.iloc[:, 1], s=10, label='y=0')
plt.legend()
plt.xlabel('x1')
plt.ylabel('x2')
plt.title('data visualization')
plt.show()

# prepare data in np.ndarray format

X_temp=X_pd.to_numpy()
X=feature_map(X_temp)
#  X: np.ndarray (100,28)  (m,n+1)
number_of_features=X.shape[1]
y = y_pd[:, np.newaxis]                     # y: np.ndarray (100,1)  (m,1)
theta = np.zeros((number_of_features, 1))          # theta: np.ndarray (28,1) (n+1,1)


# advanced optimization by scipy fmin_tnc
from scipy.optimize import fmin_tnc
def fit(self, x, y, theta):
    opt_weights = fmin_tnc(func=cost_function, x0=theta,
                  fprime=gradient,args=(x, y.flatten()))
    return opt_weights[0]

parameters = fit(fit, X, y, theta)
#print("theta by fmin_tnc:", parameters)


# predict and accuracy

h_pred=np.dot(X,np.reshape(parameters,(number_of_features,1)))
y_pred=[h_pred>0]
correct=[y_pred==y]
accuracy_fmin_tnc=np.sum(correct)/len(correct[0][0])


# sklearn LogisticRegression
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score
model = LogisticRegression(penalty='none', solver='newton-cg')
# ‘newton-cg’, ‘lbfgs’, ‘sag’ and ‘saga’ handle L2 or no penalty
model.fit(X[:,1:], np.ndarray.flatten(y))  # data array without ONE column
predicted_classes = model.predict(X[:,1:])
accuracy_sklearn = accuracy_score(y.flatten(),predicted_classes)
```

```
    par_sklearn = model.coef_
    intercept= model.intercept_
    # Print the summary of resutls
    print('theta')
    print('___fmin_tnc:', parameters)
    print('___LogistiRegression:', intercept, par_sklearn, '\n')
    print('Accuracy')
    print('___fmin_tnc:', accuracy_fmin_tnc)
    print('___LogisticRegression:',accuracy_sklearn)

    # plot decision boundary
    colors = ('red','blue','lightgreen','gray','cyan')
    cmap = ListedColormap(colors[:len(np.unique(y))])

    x1_min, x1_max = X[:,1].min()-0,X[:,1].max()+0
    x2_min, x2_max = X[:,2].min()-0,X[:,2].max()+0
    resolution=0.02
    xx1, xx2 = np.meshgrid(np.arange(x1_min, x1_max, resolution), np.arange(x2_min, x2_max, re
solution))

    temp = np.array([xx1.ravel(),xx2.ravel()]).T
    XX=feature_map(temp)

    # plot decision boundary for fmin_tnc method
    h_pred=np.dot(XX,np.reshape(parameters,(number_of_features,1)))
    h_pred=h_pred.reshape(xx1.shape)
    y_pred=np.array([h_pred>0]).reshape(xx1.shape)

    plt.contourf(xx1,xx2,y_pred, alpha=0.4, cmap=cmap)
    plt.xlim(xx1.min(),xx1.max())
    plt.ylim(xx2.min(),xx2.max())
    plt.scatter(X_admit.iloc[:, 0], X_admit.iloc[:, 1], s=10, label='Admitted')
    plt.scatter(X_not_admit.iloc[:, 0], X_not_admit.iloc[:, 1], s=10, label='Not Admitted')
    plt.title('data and decision boundaries (fmin_tnc)')
    plt.show()

    # plot decision boundary for sklearn LogisticRegression() method
    parameters_sklearn=np.c_[intercept, par_sklearn].flatten()
    h_pred=np.dot(XX,np.reshape(parameters_sklearn,(number_of_features,1)))
    h_pred=h_pred.reshape(xx1.shape)
    y_pred=np.array([h_pred>0]).reshape(xx1.shape)

    plt.contourf(xx1,xx2,y_pred, alpha=0.4, cmap=cmap)
    plt.xlim(xx1.min(),xx1.max())
    plt.ylim(xx2.min(),xx2.max())
    plt.scatter(X_admit.iloc[:, 0], X_admit.iloc[:, 1], s=10, label='Admitted')
    plt.scatter(X_not_admit.iloc[:, 0], X_not_admit.iloc[:, 1], s=10, label='Not Admitted')
    plt.title('data and decision boundaries (sklearn: LogisticRegression)')
    plt.show()

    # regularization

    lamda=1
    # advanced optimization by scipy fmin_tnc
    def fit(self, x, y, theta):
        opt_weights = fmin_tnc(func=cost_function_regularization, x0=theta,
                      fprime=gradient_regularization,args=(x, y.flatten(), lamda))
        return opt_weights[0]

    parameters = fit(fit, X, y, theta)
    #print("theta by fmin_tnc:", parameters)


    # predict and accuracy

    h_pred=np.dot(X,np.reshape(parameters,(number_of_features,1)))
    y_pred=[h_pred>0]
    correct=[y_pred==y]
    accuracy_fmin_tnc=np.sum(correct)/len(correct[0][0])


    # sklearn LogisticRegression
    from sklearn.linear_model import LogisticRegression
    from sklearn.metrics import accuracy_score
    model = LogisticRegression(penalty='l2', solver='newton-cg')
    # ‘newton-cg’, ‘lbfgs’, ‘sag’ and ‘saga’ handle L2 or no penalty
    model.fit(X[:,1:], np.ndarray.flatten(y))  # data array without ONE column
    predicted_classes = model.predict(X[:,1:])
    accuracy_sklearn = accuracy_score(y.flatten(),predicted_classes)
    par_sklearn = model.coef_
    intercept= model.intercept_
    # Print the summary of resutls
    print('theta')
    print('___fmin_tnc:', parameters)
    print('___LogistiRegression:', intercept, par_sklearn, '\n')
    print('Accuracy')
    print('___fmin_tnc:', accuracy_fmin_tnc)
    print('___LogisticRegression:',accuracy_sklearn)

    # plot decision boundary for fmin_tnc method
    h_pred=np.dot(XX,np.reshape(parameters,(number_of_features,1)))
    h_pred=h_pred.reshape(xx1.shape)
    y_pred=np.array([h_pred>0]).reshape(xx1.shape)

    plt.contourf(xx1,xx2,y_pred, alpha=0.4, cmap=cmap)
    plt.xlim(xx1.min(),xx1.max())
    plt.ylim(xx2.min(),xx2.max())
    plt.scatter(X_admit.iloc[:, 0], X_admit.iloc[:, 1], s=10, label='Admitted')
    plt.scatter(X_not_admit.iloc[:, 0], X_not_admit.iloc[:, 1], s=10, label='Not Admitted')
    plt.title('data and decision boundaries (fmin_tnc)')
    plt.show()
    # plot decision boundary for sklearn LogisticRegression() method
    parameters_sklearn=np.c_[intercept, par_sklearn].flatten()
    h_pred=np.dot(XX,np.reshape(parameters_sklearn,(number_of_features,1)))
    h_pred=h_pred.reshape(xx1.shape)
    y_pred=np.array([h_pred>0]).reshape(xx1.shape)

    plt.contourf(xx1,xx2,y_pred, alpha=0.4, cmap=cmap)
    plt.xlim(xx1.min(),xx1.max())
    plt.ylim(xx2.min(),xx2.max())
    plt.scatter(X_admit.iloc[:, 0], X_admit.iloc[:, 1], s=10, label='Admitted')
    plt.scatter(X_not_admit.iloc[:, 0], X_not_admit.iloc[:, 1], s=10, label='Not Admitted')
    plt.title('data and decision boundaries (sklearn: LogisticRegression)')
    plt.show()
```