HOMEWORK#2: LOGISTIC REGRESSION & SOFTMAX

Statistical Pattern Recognition

چکیده

گزارش تمرین جلسه ۲ – حل مسئله Classification . پیاده سازی با استفاده از loss function cross entropy

افروز راشدی اشرفی / ۴۰۰۶۲۶۹۳ - حسین توکلیان / ۴۰۰۶۲۳۵۳

آذر ۱۴۰۰

هدف: حل مسئله classification

روش اجرا: روش Logistic Regression در فضای گسسته در دو فضای باینری و چندکلاسه

ورودی: دیتاست Iris

خروجی: پارامترهای Theta، Train Accuracy، Test Accuracy، نمودار decision boundry و معادله مرز تصمیم

زبان برنامه: Python

مدل Logistic Regression، یک classifier یادگیری نظارتی است که فیچر های واقعی را از ورودی استخراج کرده، هرکدام را در یک وزن ضرب می کند، همه را با هم جمع می کند و سپس این حاصل جمع را از طریق یک تابع sigmoid برای تولید احتمال استفاده می کند. همچنین برای تصمیم گیری، به یک threshold (آستانه) نیاز دارد.

در این تمرین، ما مسئله Classification را با در نظر گرفتن مسئله به صورت های باینری و چندکلاسه و با استفاده از دیتاست iris حل میکنیم.

بخش اول) Binary Classification

دیتاست ما دارای ۴ فیچر است که برای بخش باینری، تنها از کلاسهای ۱ و ۲ در این دیتاست استفاده شده است. همچنین کلاس ها از ۳ به ۲ تقلیل یافته اند که در نتیجه مسئله ما یک مسئله ۲ کلاسه و ۲ بعدی است.

۸۰ درصد داده ها برای train و ۲۰ درصد برای test استفاده شده اند.

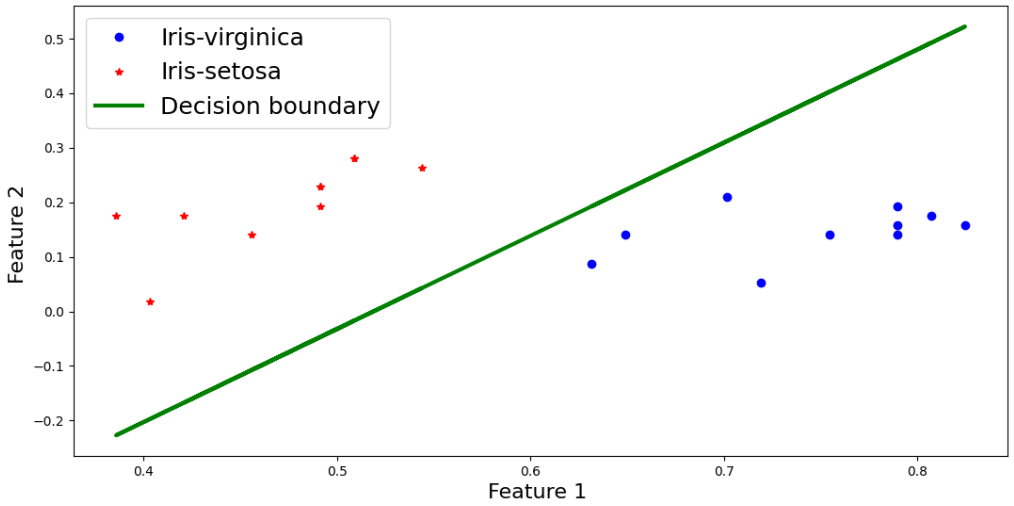
روش اجرا:

پارامترهای مدل یعنی w و b، باید learn شوند. در یادگیری نظارتی، مقدار دقیق y را می دانیم. بنابراین جوری باید پارامترها را learn کنیم که y حاصل (یعنی y^) به مقدار واقعی نزدیکتر باشد. برای این منظور، به ۲ مولفه نیاز است: loss function و الگوریتم optimization. در این تمرین برای پیاده سازی loss function از Cross-entropy loss و برای الگوریتم بهینه سازی از stochastic gradient descent استفاده شده است.

تابع زیر SGDforLR، محاسبات ما را برای پیاده سازی دسته بندی باینری انجام میدهد.

```
def y_gradient_desc(x):
    y_gd = []
    for i in range(x.shape[0]):
        a = e**(np.inner(x[i].reshape(x[i].shape[0],1).T,
gradient_desc_w[1:].T)+gradient_desc_w[0])
        b = (1.+e**(np.inner(x[i].reshape(x[i].shape[0],1).T,
gradient_desc_w[1:].T)+gradient_desc_w[0]))
        y_gd.append((a/b)[0])
    y_gd = np.round(y_gd)
    return y_gd
```

خروجی های الگوریتم باینری:



################## Calculated Ws ######################

Gradient Descent Solution, thetas: [[3.83635586] [-7.3934824] [4.32727652]]

############################

Decision boundary formula:

-(3.8363558602575525 + 3.8363558602575525*X1) / (4.327276523355108)

############################

Train accuracy: 0.9873417721518988

Test accuracy: 1.0

بخش دوم) Multiclass Classification

در این بخش، از تمام داده های دیتاست (۳ کلاس و ۴ فیچر) استفاده شده است. در اینجا ما می خواهیم مرز بین چند کلاس را مشخص کنیم. برای کلاس بندی دو تکنیک ovo و ova در ادامه توضیح داده می شوند:

تکنیک اول) One-vs-One

یکی از تکنیک های حل مسئله classification، تکنیک ovo است. در این روش ما هر کلاس را با یک کلاس دیگر مقایسه می کنیم.

در این روش اگر n کلاس داشته باشیم به تعداد n(n-1)/2 classifier نیاز داریم.

کد زیر، تکنیک ovo را برای ما پیاده سازی می کند

```
def SGDforLR(x, y, alpha, maxIter):
    # Stochastic gradient descent - logistic regression with static
learning rate
    m, n = x.shape
    w_0 = np.random.rand(n).reshape(n, 1)
    w_1 = np.zeros((n, 1))
    it = 0
    while (it <= maxIter):
        for i in range(m):
            w_0 = w_1
            a = e**(np.inner(x[i][1:].reshape(x[i][1:].shape[0], 1).T,
w_0[1:].T)+w_0[0])
            b = (1.+e**(np.inner(x[i][1:].reshape(x[i][1:].shape[0], 1).T,
w_0[1:].T)+w_0[0]))
            logisticFunction = (a/b)
            error = (y[i]-logisticFunction)
            # Update weights using SGD based on the cross entropy
            w_1 = w_0 + (alpha*error[0]*x[i]).T.reshape(w_0.shape)
#             print(w_1.T)
            i += 1

        it += 1
    return w_1

# Run SGD
nClass = 3
ovoModels = {}
it = -1
for i in range(nClass):
    for j in range(i+1, nClass):
        it += 1
        if i != j:
            ovo = y_trn[np.any(np.concatenate(
                (y_trn == i, y_trn == j), axis=1), axis=1), :]
            ovo[ovo == i] = 0
            ovo[ovo == j] = 1
            XTrainBiasAdded_ovo = XTrainBiasAdded[np.any(
                np.concatenate((y_trn == i, y_trn == j), axis=1), axis=1),
:]
            gradient_desc_w = SGDforLR(
                XTrainBiasAdded_ovo, ovo, alpha=0.003, maxIter=1000)
            ovoModels[it] = gradient_desc_w

            # Show w
            print('################ Calculated Ws for classes: ', i+1, '
and ', j+1, '################')
            print('Gradient Descent Solution, thetas: ',
                  gradient_desc_w.T, '\n', '----------')
```

خروجی الگوریتم One-vs-One:

################## Calculated Ws for classes: 1 and 2 ##################

Gradient Descent Solution, thetas: [[3.53658963 -0.91280483 2.63875412 -8.20151019 -3.7572857]]

################## Calculated Ws for classes: 1 and 3 ##################

Gradient Descent Solution, thetas: [[3.84952609 -0.14647792 0.46536996 -5.24367555 -3.17018849]]

################## Calculated Ws for classes: 2 and 3 ##################

Gradient Descent Solution, thetas: [[-2.80247485 1.20564456 -3.4802312 8.71812129 3.57893361]]

Train accuracy: 0.9495798319327731

Test accuracy: 1.0

تکنیک دوم) One-vs-All

در این روش ما هر کلاس را به تنهایی در برابر سایر کلاسها قرار می دهیم و مقایسه را انجام می دهیم.

اگر n کلاس داشته باشیم، به n کلاسیفایر نیاز داریم.

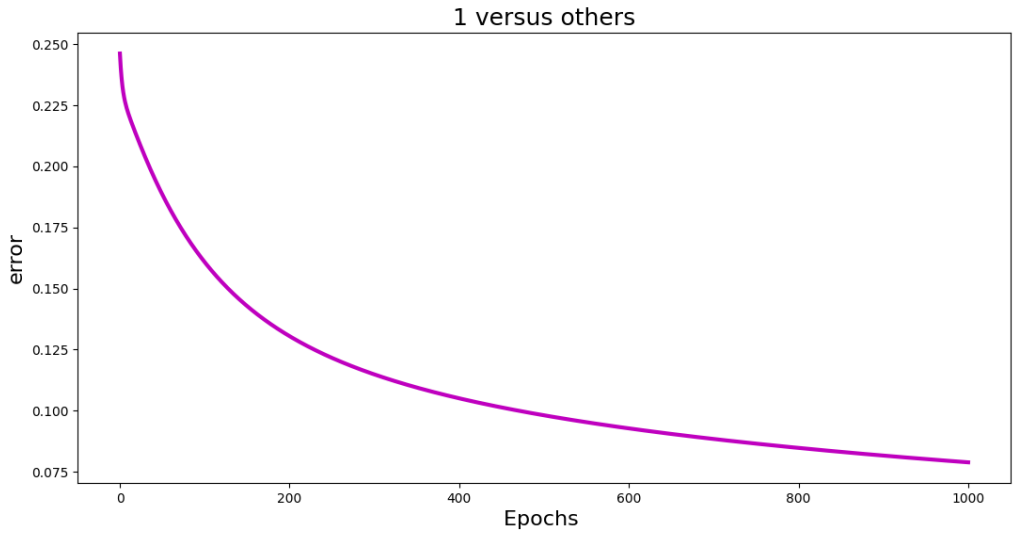
کد زیر، تکنیک ova را برای ما پیاده سازی می کند.

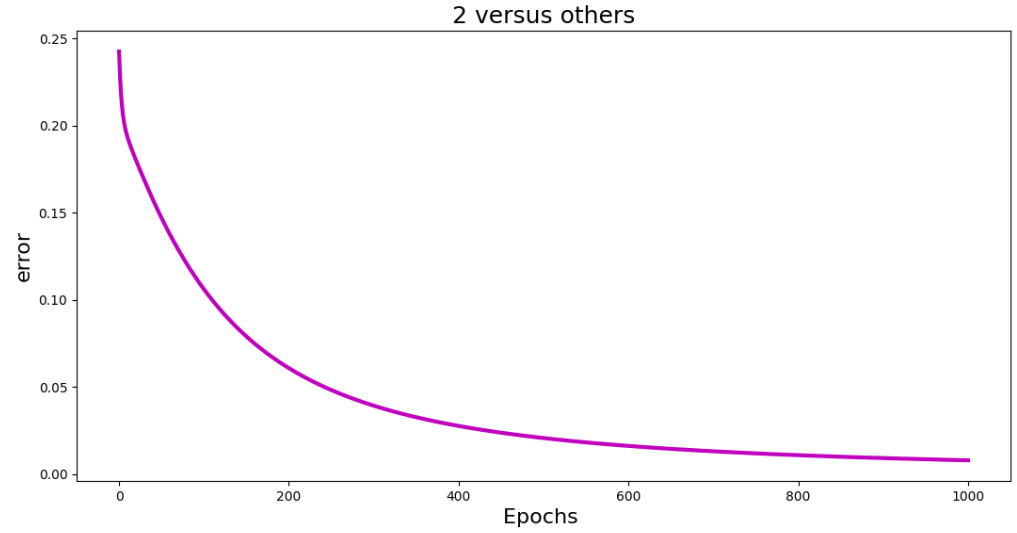
```
def SGDforLR(x, y, alpha, MaxIter):
    # Stochastic gradient descent - logistic regression with static
learning rate
    m, n = x.shape
    w_0 = np.random.rand(n).reshape(n, 1)
    w_1 = np.zeros((n, 1))
    i = 0
    it = 0
    errors = []
    while (it <= MaxIter):
        Out = []
        for i in range(m):
            w_0 = w_1
            A = e**(np.inner(x[i][1:].reshape(x[i][1:].shape[0], 1).T,
w_0[1:].T)+w_0[0])
            B = (1+e**(np.inner(x[i][1:].reshape(x[i][1:].shape[0], 1).T,
w_0[1:].T)+w_0[0]))
            LogisticFunction = A/B
            error = (y[i]-LogisticFunction)
            # Update weights using SGD based on the cross entropy
            w_1 = w_0 + (alpha*error[0]*x[i]).T.reshape(w_0.shape)
            Out.append(LogisticFunction[0][0])
            # print(w_1.T)
            i += 1
        Out = np.array(Out)
        MSE = np.mean((y-Out.reshape(Out.shape[0], 1))**2)
        errors.append(MSE)
```

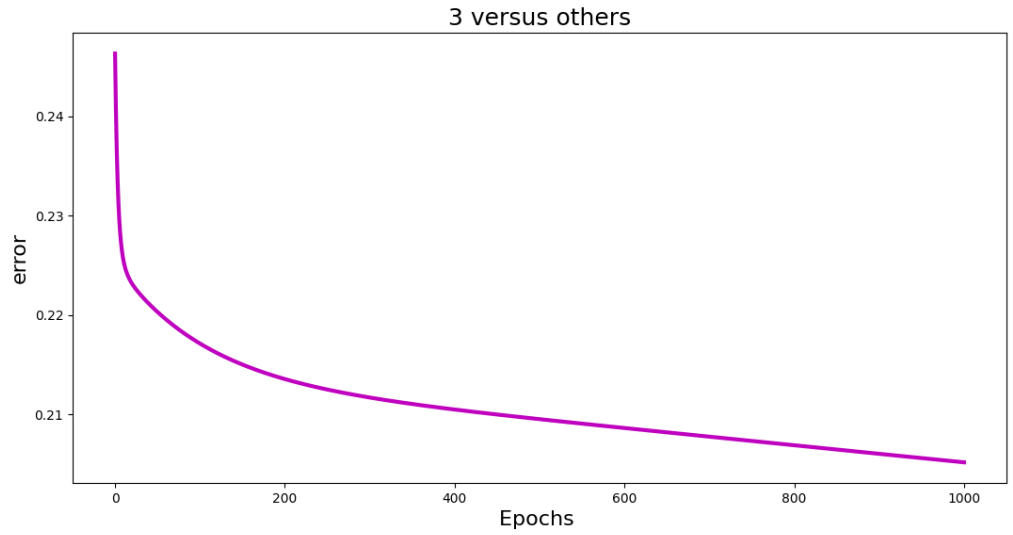
```
        it += 1
    return w_1, np.array(errors)


nClass = 3
ovaModels = []
for i in range(nClass):
    ova = np.zeros(y_trn.shape)
    ova[y_trn == i] = 1
    gradient_desc_w, errors = SGDforLR(
        x_trnBiasAdded, ova, alpha=0.003, MaxIter=1000)
    ovaModels.append(gradient_desc_w)
```

خروجی الگوریتم One-vs-All:







################## Calculated Ws for class: 1 ##################

Gradient Descent Solution, thetas:

[[-5.08329552 0.06281486 -1.80382429 7.71211909 4.23751185]]

################## Calculated Ws for class: 2 ##################

Gradient Descent Solution, thetas:

[[3.19631204 -1.35757029 3.52089027 -9.72184284 -4.15325014]]

################## Calculated Ws for class: 3 ##################

Gradient Descent Solution, thetas:

[[-3.98297791e-04 -3.80364051e-01 -3.23010448e+00 1.80842545e+00
-7.79185315e-02]]

Train accuracy: 0.907563025210084

Test accuracy: 1.0

بخش سوم) Softmax

سافتمکس یا multinomial logistic regression، روشی برای دسته بندی چندکلاسه است. در این روش، y هدف، متغیری است که بیش از ۲ کلاس دارد. در اینجا می خواهیم احتمال بودن y از هر کلاس بالقوه را تشخیص دهیم.

این multinomial logistic classifier از تعمیم تابع سیگموید، به نام softmax استفاده می کند. تابع سافتمکس وکتور z را به ازای k مقدار می گیرد و آنها را به توزیع احتمال نگاشت می کند (هر داده در بازه صفر و یک است و جمع همه مقادیر ۱ می شود).

تابع softmax مانند سیگموید یک تابع نمایی است.

در multinomial logistic regression، برای هر کلاس، وزن جداگانه ای در نظر می گیریم و برای learn کردن از تعمیم cross-entropy loss برای باینری از ۲ کلاسه به چندکلاسه استفاده میکنیم.

بنابراین loss function برای یک سمپل x، جمع لگاریتم K کلاس است که هرکدام با احتمال کلاس واقعی (y_k) وزن دار شده اند. از آنجایی که فقط یک کلاس، کلاس صحیح است وکتور y فقط برای این مقدار k یک می شود. بنابراین آن را one-hot vector می نامند.

الگوریتم زیر، softmax را اجرا می کند

```
def SGDforSoftmaxLR(x, y, alpha, maxIter, nClass):
    # Stochastic gradient descent for softmax logistic regression with
static learning rate
    m, n = x.shape
    w_0 = np.random.rand(n, nClass)
    w_1 = np.zeros((n, nClass))
    for classidx in range(nClass):
        it = 0
        while (it <= maxIter):
            for i in range(m):
                temp1 = w_1
                w_0 = temp1
                A = e**(np.inner(x[i][1:].reshape(x[i][1:].shape[0], 1).T,
w_0.T[classidx].reshape(
                    w_0.T[classidx].shape[0],
1)[1:].T)+w_0.T[classidx].reshape(w_0.T[classidx].shape[0], 1)[0])

                B = 0
                for ii in range(nClass):
                    temp = B =
(e**(np.inner(x[i][1:].reshape(x[i][1:].shape[0], 1).T, w_0.T[ii].reshape(
                        w_0.T[ii].shape[0],
1)[1:].T)+w_0.T[ii].reshape(w_0.T[ii].shape[0], 1)[0]))
                    B += temp

                probability = (A/B)[0]

                temp = w_0[:, classidx].reshape(w_0.shape[0], 1) - alpha*(-
1 * ((logicFunction(y, classidx)[
                    i]-probability)*x[i])).reshape(w_0.shape[0], 1)   #
.T.reshape(w_0.shape)    # Update weights using SGD
                for qq in range(n):
                    w_1[qq][classidx] = temp[qq][0]
```

```
            it += 1
    return w_1

# output Gradient_descent
nClass = 3
for it in range(nClass):
    y_softmax = []
    Gradient_descent_w = gradient_desc_w_Mat[:, it]
    for i in range(x_trn.shape[0]):
        A = e**(np.inner(x_trn[i].reshape(x_trn[i].shape[0], 1).T,
                Gradient_descent_w[1:].T)+Gradient_descent_w[0])
        B = 0
        for ii in range(nClass):
            Gradient_descent_w_others = gradient_desc_w_Mat[:, ii]
            temp = e**(np.inner(x_trn[i].reshape(x_trn[i].shape[0], 1).T,

Gradient_descent_w_others[1:].T)+Gradient_descent_w_others[0])
            B += temp
        y_softmax.append((A/B)[0])
    if it == 0:
        y_softmax_train = np.array(y_softmax).reshape(
            np.array(y_softmax).shape[0], 1)
    else:
        y_softmax_train = np.concatenate((y_softmax_train, np.array(
            y_softmax).reshape(np.array(y_softmax).shape[0], 1)), axis=1)
```

خروجی softmax:

################## Calculated Ws ######################

Gradient Descent Solution, thetas: [[-3.62431598 2.00688934 -156.156]

[-2.35476043 -1.45777072 -111.39333333]

[-1.77696518 2.51014338 -70.532]

[7.43829796 -8.33792275 -53.79733333]

[5.40598709 -3.73551832 -15.81066667]]

Train accuracy: 0.6638655462184874

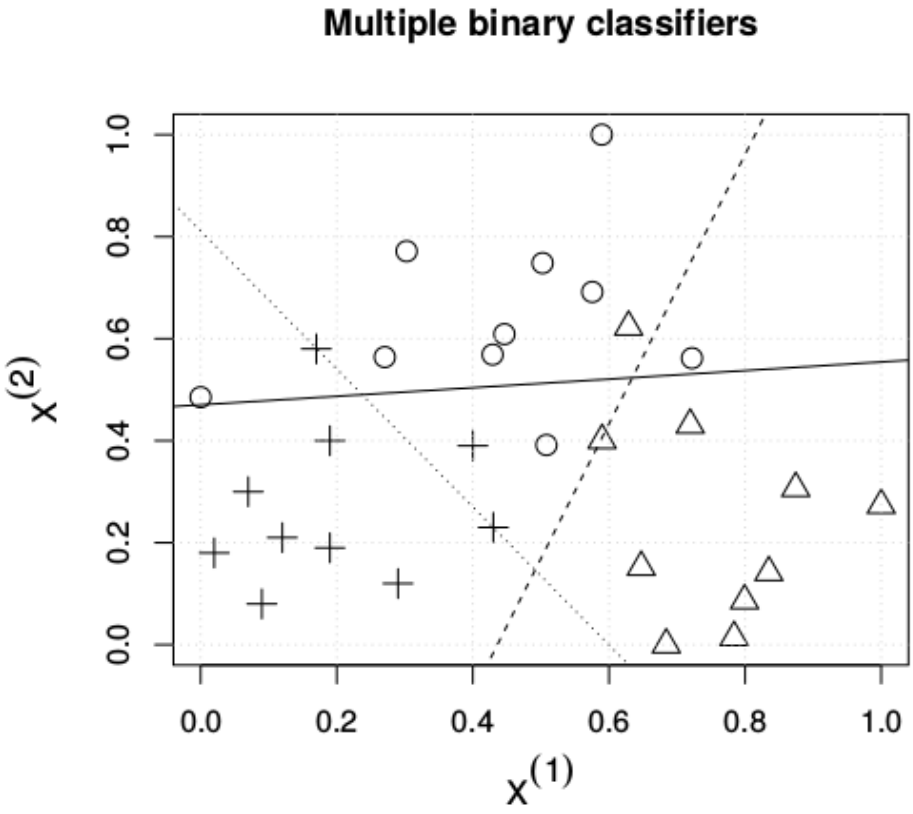
Test accuracy: 0.6666666666666666

Conclusion:[1]

Generally speaking, I think that adapt multiple binary classifiers is not always the best way to deal with a multi-class classification problem.

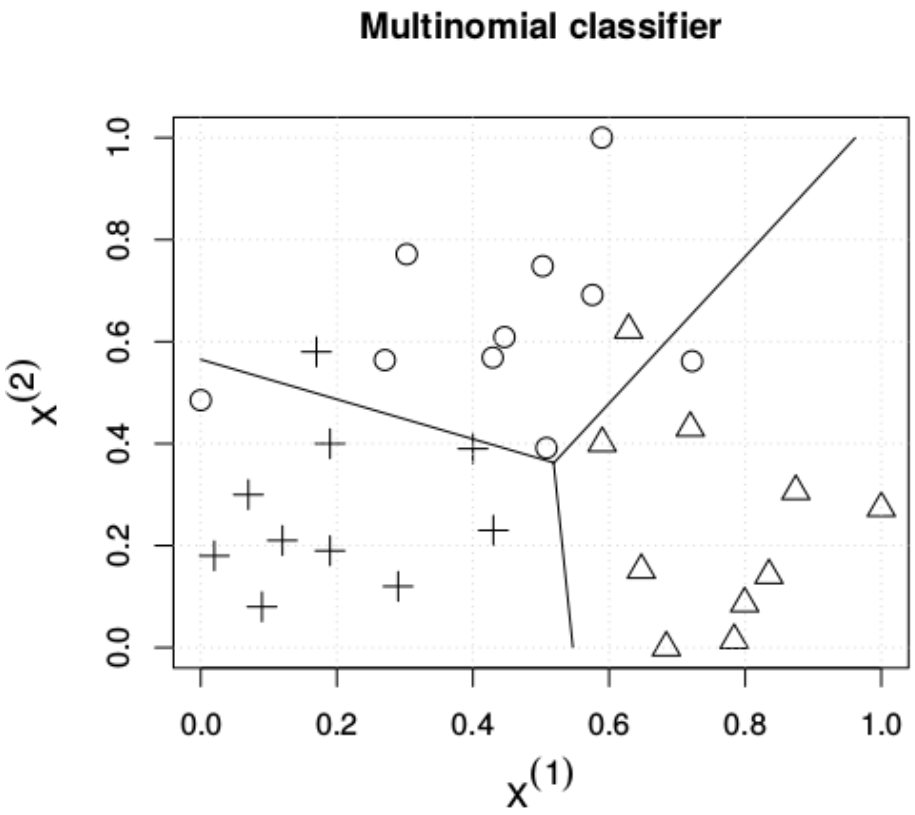
If your dataset can be learned approximately with linear hypothesis, it could be interesting to use a multinomial logistic regression (also known as Maximum Entropy classifier).

For instance, consider a dataset with three classes and two features $(x_{(1)},x_{(2)})$(x(1),x(2)). In an "one-to-rest" strategy, one could build 3 independent classifiers and, for an unseen instance, choose the class for which the confidence is maximized. In this case, the decision boundaries may be drawn as following:

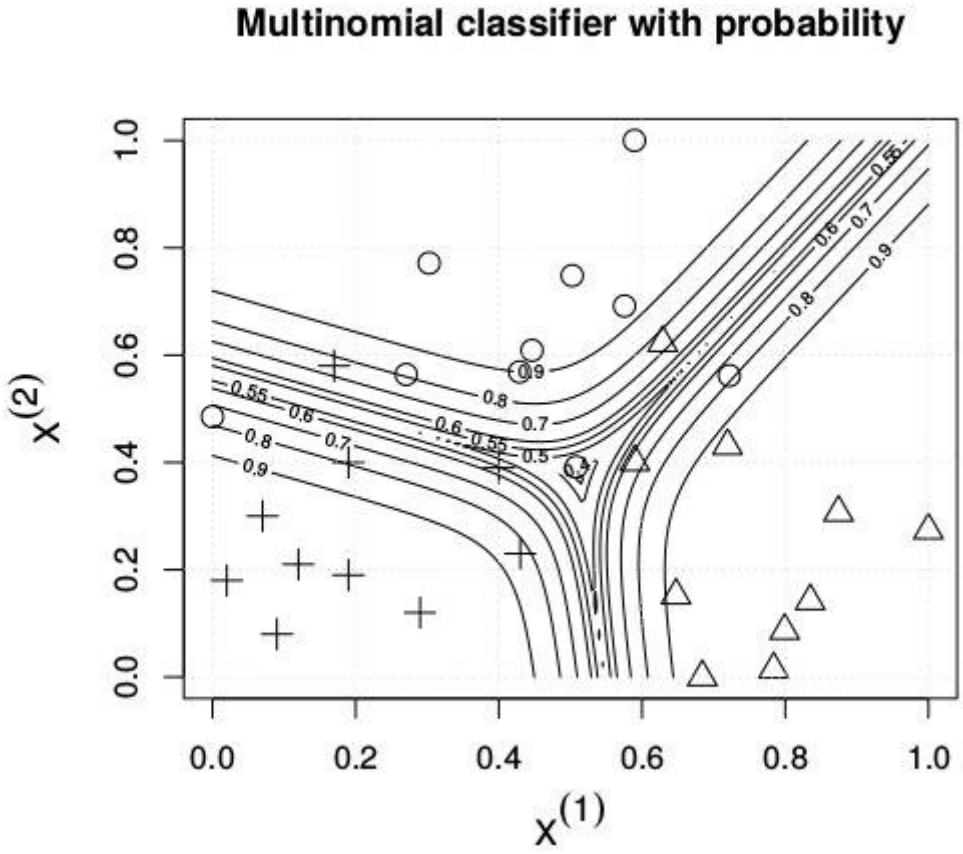


Conversely, another strategy is to use a multinomial classifier which learns directly all the classes. In this way, the parameters of each class are estimated interdependently and the model built may be more robust against outliers:

[1] https://www.quora.com/In-multi-class-classification-what-are-pros-and-cons-of-One-to-Rest-and-One-to-One



Moreover, in this case, you have a global distribution where, for an instance, the sum of all the probabilities is equal to 1:



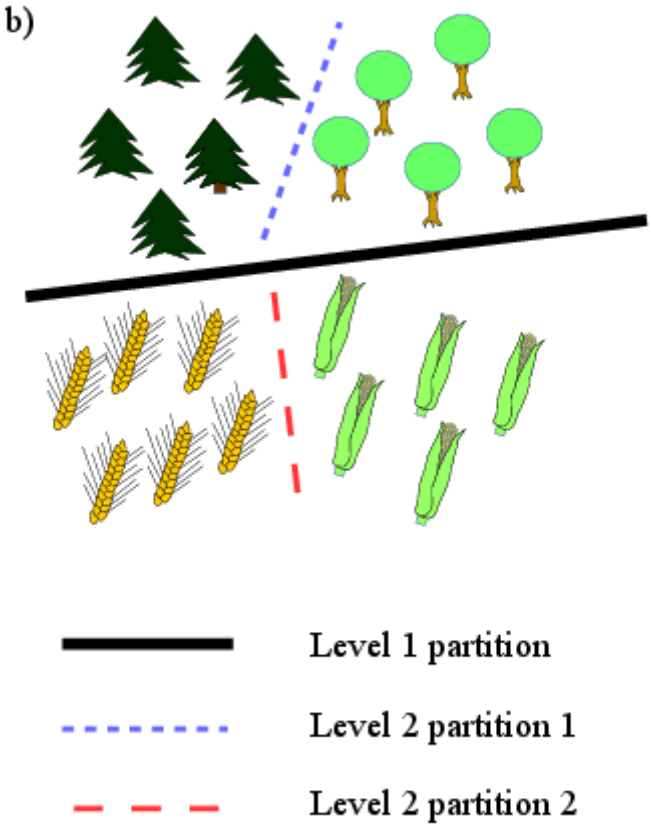
First I'd like to discuss the multiple binary classifiers vs one multinomial classifier part.

In general this choice depends on how your data relates to the classes. If your data can only belong exclussively to one class then the multinomial classifier is the right choice. A logistic regression softmax classifier is the typical example. The MNIST digits database is a good example of this as a digit is either a 0,1,2,.... or a 9. It can't be part 0 and part 5.

When your data can belong to more than one class with different degrees then it is better to train a binary classifier for each class. For example if you are classifying music in genres a song can be mostly "pop" but have some of "country". In these cases multiple binary classifiers will work better.

Now if you want to use multiple binary classifiers you have the 1 vs 1 and 1 vs all choice as your question states. Then I'm going to agree with Eren Golge the 1 vs 1 method usually works slightly better than 1 vs all but is severely limited by the number of classes you have. If you only have a few classes then no problem. But for many classes 1 vs all is the right choice.

When generalizing binary to multi-class classification, the best method to use will be highly specific to the problem. Here is a paper I wrote specifically addressing this point: [1404.4095] Multi-borders classification
It describes software for designing a multi-class statistical classification model through composition of binary classifiers. One problem on which I'm testing this software illustrates this point very nicely. One-against-all works terribly. One-against-one works OK but the best method by far is partitioning all the classes between each pair of adjacent classes. Why? Because it is a continuum regression problem that I've discretized into eight classes. The classes have an ordering and it is better to preserve this ordering: keeping contiguous ranges contiguous.



For another example that illustrates this point well, consider a land classification problem consisting of four surface types: coniferous forest, deciduous forest, corn field and wheat field. Surely the best method for partitioning these is hierarchical? First, discriminate between forest and field. Then, if forest, discriminate between deciduous and coniferous, if field, discriminate between corn and wheat

First thing is the number of models you need to train. If you have N number of classes, one-to-one approach needs N^2-N models, t is pretty high if your N is 1000. One to rest is better in that sense and from my own personal experience I can say that they have very similar performance values if your features are good. Therefore in general One-To-Rest is the choice.

However, if you have some sort of special considerations between classes due to unbalanced number of instances or pair specific features (for instance feature A might be useful to discern Class 1 from Class 2 but not so for other classes). In such cases, you can use one-to-one approach.