Lecture 4: Logistic Regression

DATA 602 @ UMBC

Masoud Soroush

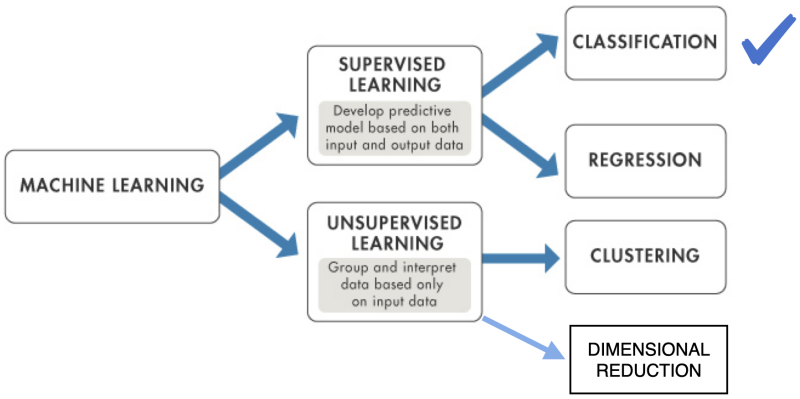
September 19, 2022

1 Classification Algorithms

In today's lecture[1], we introduce the very first classification algorithm, namely the Logistic Regression. During the course of the next few weeks, we will introduce a wide range of classification methods that are commonly used in machine learning. Logistic regression which is the oldest classification algorithm is used to predict the value of a categorical target variable. In addition, in today's lecture, we will introduce the relevant metrics for assessing the performance of classification algorithms.

1.1 Background

Classification algorithms belong to the realm of *supervised* machine learning. As we saw in previous weeks, the target variable for regression methods was a *continuous variable*. In contrast, when the target variable is categorical, we need to apply a different type of algorithm to predict the target. The process of predicting a categorical target variable in machine learning is called **classification**.



[1] This product is a property of UMBC, and no distribution is allowed. These notes are solely for your own use, as a registered student in this class.

There are a wide range of classification techniques in machine learning. The very first classification technique (that we will discuss today) is logistic regression. Note that the term regression in the title of this classification algorithm shouldn't mislead you to think that logistic regression is a type of regression for predicting a continuous target! *Logistic regression is not a regression algorithm! It is rather a classification algorithm!* Other classification algorithms that will be studied during the course of the next few weeks are *Support Vector Machine*, *Naive Bayes*, *Decision Trees*, and some ensemble learning methods such as *Random Forest Classifier*.

1.2 Logistic Regression Formulation

Before introducing the formulation of logistic regression, we distinguish two types of classification: the binary and the multinomial classifications.

Binary Classification: When the number of possible outcomes for a categorical target variable is exactly two, we refer to the situation as a *binary classification*. Some examples of binary classification problems are:

- target variable: gender (possible outcomes: male, female)
- target variable: party affiliation in US (Possible outcomes: Democratic, Republican)
- target variable: a yes/no survey question (possible outcomes: Yes/Agree, No/Disagree)
- target variable: health condition (possible outcomes: Healthy, Not Healthy)

Multinomial Classification: When the number of possible outcomes for a categorical target variable is greater than two, we refer to the situation as a *multinomial classification*. Some examples of multinomial classification problems are:

- target variable: color (possible outcomes: Blue, Red, Green)
- target variable: car types (possible outcomes: SUV, Sedan, Hatchback, Convertible, Coupe)
- target variable: age category (possible outcomes: Children, Youth, Adults, Seniors)
- target variable: house type in US (possible outcomes: Single Family, Condominium, Townhouse, Multi-Family)

We will formulate logistic regression for both binary and multinomial classification problems.

1.2.1 Binary Logistic Regression

In this section, we formulate the binary logistic regression. To set the stage, assume that we have *d continuous features* collectively represented by $\vec{x} = (x_1, x_2, \cdots, x_d)$. The target variable y is a categorical variable with exactly two possible outcomes. It is convenient to label the two outcomes as **Failure** and **Success** associated with $y = 0$ and $y = 1$, respectively.

$$\begin{cases} y = 0 : & \text{Failure} \\ \\ y = 1 : & \text{Success} \end{cases} \tag{1}$$

The above labeling is merely a choice, and you can change the labeling without affecting the predictions of the model. Suppose we have recorded the values of features and the target for n observations.

Since the target variable is categorical, we cannot assume a relation of the form $\mathbb{Y} = \mathbb{X}\,\omega^{\mathsf{T}}$ (as in the case of linear regression) between the target and features. Such a relation would be meaningless as one equates a categorical quantity with a continuous quantity!. We have to take a different approach. Logistic regression takes a **probabilistic approach** to cure this problem. We notice that although y is a categorical variable, *the probability for y to assume a certain class* is continuous! We define

$$\begin{aligned} P(y^{(i)}|\vec{x}^{(i)}) : &\text{ probability that the target variable takes value } y^{(i)} \text{ for the } i\text{-th} \\ &\text{observation provided that the features are } \vec{x}^{(i)} = (x_1^{(i)}, x_2^{(i)}, \cdots, x_d^{(i)}) \,. \end{aligned} \tag{3}$$

Note that the probability in (3) is an instance of *conditional probability*. The above probability only concerns the i-th observation. To cover the whole dataset, we need to consider all n observations. We define the total probability function $\mathscr{P}$ by

$$\mathscr{P} = P(y^{(1)}|\vec{x}^{(1)})P(y^{(2)}|\vec{x}^{(2)})\cdots P(y^{(n)}|\vec{x}^{(n)}) = \prod_{i=1}^{n} P(y^{(i)}|\vec{x}^{(i)}) \,. \tag{4}$$

Important Note: In defining the total probability function, we assume that all n observations of the dataset are **independent** of one another. Recall from your statistics course that if A and B are independent events, then the probability for the event A and B is given by $P(A \cap B) = P(A) \cdot P(B)$. That is why we multiply all the n probabilities in equation (4) associated with the n (independent) observations in the dataset. If you deal with a case where the observations are not independent of one another (*i.e.* the outcome of an earlier observation influences the outcomes of later observations), you cannot use equation (4) for the total probability.

Now that we have the total probability function (4), we should maximize it in order to find the predictions of the model for the target variable y. However, before we get into the maximization of $\mathscr{P}$, we need to decide how we model the individual probabilities $P(y^{(i)}|\vec{x}^{(i)})$.

Question: Since $P(y^{(i)}|\vec{x}^{(i)})$ is a continuous quantity, can we model it via a linear relation (*i.e.* $P(y^{(i)}|\vec{x}^{(i)}) = \omega_0 + \omega_1 x_1^{(i)} + \cdots + \omega_d x_d^{(i)}$)?

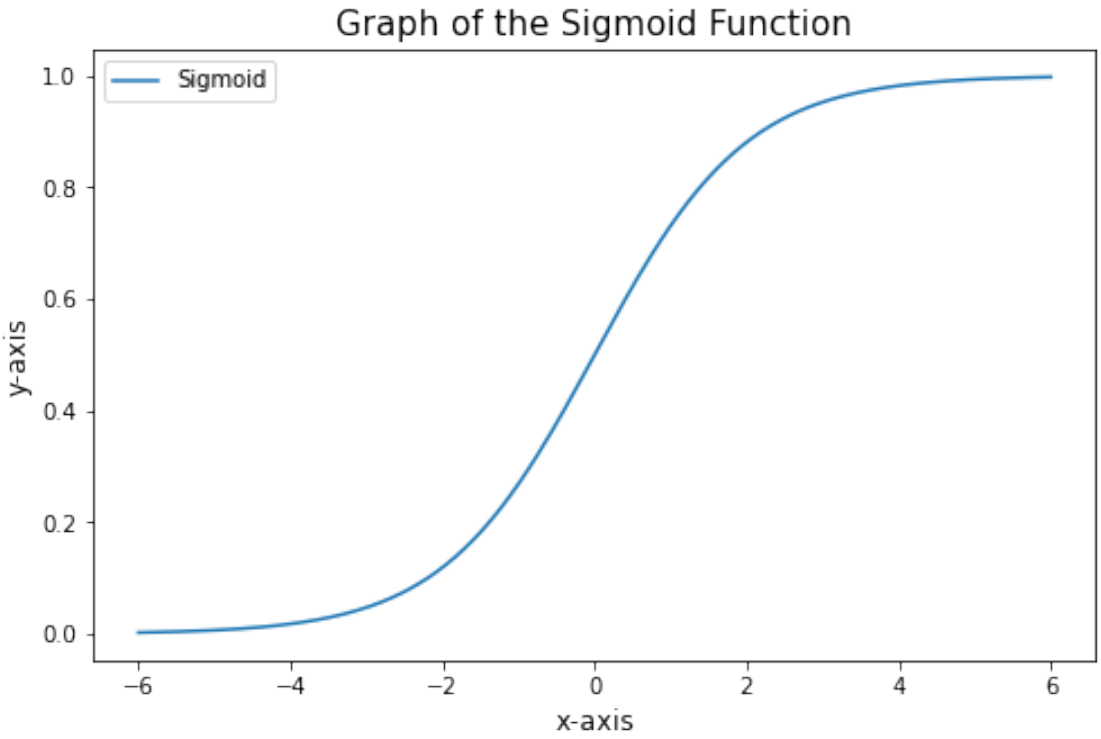
Answer: No! The linear (and polynomial) function is unbounded (*i.e.* the value of the function can vary from $-\infty$ to $+\infty$ when at least one of the weights $\omega_i \neq 0$), whereas $P(y^{(i)}|\vec{x}^{(i)})$ is a probability, and hence it belongs to the interval $(0, 1)$! The probability function cannot assume any negative values of any positive values greater than 1.

To address the above problem, we need to model the probability function $P(y^{(i)}|\vec{x}^{(i)})$ with a *bounded function*. In logistic regression, the *sigmoid function* is chosen to take care of this job. The **sigmoid function** σ is defined by

$$\begin{aligned} &\sigma : \mathbb{R} \to (0, 1) \\ &\sigma(z) = \frac{1}{1 + e^{-z}} \,. \end{aligned} \tag{5}$$

Notice that the sigmoid function (5) maps the entire real line $\mathbb{R} = (-\infty, +\infty)$ to the interval $(0, 1)$. To obtain a better sense about the sigmoid function, let us plot the function in (5) and find its graph. We can easily accomplish this task in python:

[1]:
```
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns

def sigmoid(z):                    # Define sigmoid function
    return 1/(1 + np.exp(-z))

x_vals = np.linspace(-6, 6, 200)
y_vals = sigmoid(x_vals)

sig_plot = sns.lineplot(x=x_vals, y=y_vals, label='Sigmoid')
sig_plot.set_xlabel("x-axis", fontsize=12)
sig_plot.set_ylabel("y-axis", fontsize=12)
sig_plot.set_title("Graph of the Sigmoid Function", fontsize=15)
plt.gcf().set_size_inches(8, 5)
plt.show()
```



As is clear from the above graph, sigmoid function σ is a continuous and smooth bounded function that varies in the interval $(0, 1)$ (This is exactly what we need for a probability function). It is important to note that the sigmoid function is not the only function that can be modeled for the

probability function. One can choose a different bounded function for this purpose. However, the sigmoid function is a particularly simple choice for this purpose.

We can now define a linear function z

$$z^{(i)} = \omega_0 + \omega_1 x_1^{(i)} + \cdots + \omega_d x_d^{(i)} = \omega_0 + \vec{\omega} \cdot \vec{x}^{(i)} \,, \tag{6}$$

where ω_0 as usual represents the bias term and the weight vector $\vec{\omega} = (\omega_1, \omega_2, \cdots, \omega_d)$ contains the weights that correlate with the features $\vec{x} = (x_1, x_2, \cdots, x_d)$. In order to fulfill the boundedness property of the probability function $P(y^{(i)}|\vec{x}^{(i)})$, we equate $\sigma(z^{(i)})$ with the probability to observe success (*i.e.* $y^{(i)} = 1$) for the i-th observation. Since success and failure are two disjoint outcomes, the probability for observing failure (*i.e.* $y^{(i)} = 0$) for the i-th observation will be $1 - \sigma(z^{(i)})$. In summary

$$P(y^{(i)}|\vec{x}^{(i)}) = \begin{cases} \sigma(z^{(i)}) \,, & y^{(i)} = 1 \\ \\ 1 - \sigma(z^{(i)}) \,, & y^{(i)} = 0 \end{cases} \,. \tag{7}$$

Let us denote the probability of finding success in the i-th observation by $p(z^{(i)}) = P(y^{(i)} = 1|\vec{x}^{(i)})$. Then, the first case of (7) implies that $p(z^{(i)}) = \dfrac{1}{1 + e^{-z^{(i)}}}$. One can then show that

$$e^{z^{(i)}} = e^{\omega_0 + \vec{\omega} \cdot \vec{x}^{(i)}} = \frac{p(z^{(i)})}{1 - p(z^{(i)})} \,. \tag{8}$$

The quantity on the right hand side of (8) is called the **odds** for the i-th observation. Note that odds belong to the interval $(0, \infty)$. Values of odds close to zero indicate a low probability of success for the i-th observation. On the other hand, large values of odds indicate a high probability of success for the i-th observation. Odds introduce a measure for the probability of success for each individual observation of the dataset. *Note that each observation has its own odds*. The logarithm of odds for the i-th observation is known as the **log-odds** or **logit**:

$$z^{(i)} = \omega_0 + \vec{\omega} \cdot \vec{x}^{(i)} = \log\left(\frac{p(z^{(i)})}{1 - p(z^{(i)})}\right) \,, \tag{9}$$

and as is clear from equation (9), log-odds can vary from $-\infty$ to $+\infty$.

Equivalently, we can combine the two branches of (7) into one single expression. Using a simple trick, this can be done and both branches of (7) can be simultaneously described as follows:

$$P(y^{(i)}|\vec{x}^{(i)}) = \sigma(z^{(i)})^{y^{(i)}} \, (1 - \sigma(z^{(i)}))^{1 - y^{(i)}} \,. \tag{10}$$

Note that when $y^{(i)} = 1$ (*i.e.* success takes place), then $1 - y^{(i)} = 0$, and hence $P(y^{(i)}|\vec{x}^{(i)}) = \sigma(z^{(i)})$. On the other hand, when $y^{(i)} = 0$ (*i.e.* failure takes place), then $1 - y^{(i)} = 1$ and $P(y^{(i)}|\vec{x}^{(i)}) = 1 - \sigma(z^{(i)})$. As usual, to present the total probability function $\mathscr{P}$, we need to multiply the probabilities associated with all n observations of the dataset. Hence, we arrive at

$$\boxed{\mathscr{P} = \prod_{i=1}^{n} \sigma(z^{(i)})^{y^{(i)}} \left(1 - \sigma(z^{(i)})\right)^{1-y^{(i)}}} \,. \tag{11}$$

Goal: We aim to find the **optimal values** for the weights ω_0 and $\vec{\omega}$ that maximize the total probability function (11).

In maximizing the total probability function, we encounter a technical difficulty. Recall from lecture 1 that in order to maximize a function, you need to take the derivatives with regard to parameters (in this case $\omega_0, \omega_1, \cdots, \omega_d$) and set them to zero. From the product rule for derivatives, you recall that taking the derivative of a product is a tedious task! On the other hand, taking the derivative of a sum is very straightforward: the derivative of the sum is equal to the sum of individual derivatives! Therefore computationally, it would be highly desirable to convert the product in (11) into a sum:

$$\prod_{i=1}^{n} \longrightarrow \sum_{i=1}^{n} \,. \tag{12}$$

We can easily do that by defining the **likelihood function** to be the logarithm of the total probability function

$$\ell(\omega_0, \vec{\omega}) = \log(\mathscr{P}) \,. \tag{13}$$

The logarithm on the right hand side of (13) automatically converts the product into a sum due to the property $\log(ab) = \log(a) + \log(b)$. Since logarithm is a monotonically increasing function, maximization of the total probability function results in the maximization of the likelihood function. In machine learning, however, people prefer to minimize a cost function (rather than maximizing a function). We can easily fulfill the desire by introducing the cost function to be

$$J(\omega_0, \vec{\omega}) = -\ell(\omega_0, \vec{\omega}) = -\sum_{i=1}^{n} \left[y^{(i)} \log(\sigma(z^{(i)})) + (1 - y^{(i)}) \log(1 - \sigma(z^{(i)})) \right] . \tag{14}$$

Because of the minus sign on the right hand side of (14), the maxima of the likelihood function correspond to the minima of the cost function of the logistic regression. Note that $J(\omega_0, \vec{\omega})$ is referred to as the **cost function of the logistic regression algorithm**. To minimize $J(\omega_0, \vec{\omega})$, we set $\dfrac{\partial J(\omega_0, \vec{\omega})}{\partial \omega_j} = 0$ which, after some simple algebraic manipulations, lead to

$$\sum_{i=1}^{n} \left(y^{(i)} - \sigma(z^{(i)})\right) x_j^{(i)} = 0 \,, \tag{15}$$

for $j = 0, 1, \cdots, d$ with $x_0^{(i)} = 1$. Note that (15) represents $d + 1$ equations, as j is a free index ranging from 0 to d (Do not take (15) as a single equation!). This system of equations need to be solved in order to determine the optimal values for ω_0 and $\vec{\omega}$. However, unlike the regression case, this cannot be done analytically! scikit-learn internally solves this system of equations using a gradient descent approach through the substitution $\vec{\omega} \longrightarrow \vec{\omega} - \eta \, \vec{\nabla} J(\vec{\omega})$.

1.2.2 Multinomial Logistic Regression

In this section, we formulate the logistic regression algorithm for the multinomial classification problems. In the literature, the multinomial logistic regression is sometimes known as the *Softmax Regression* as well. The first observation to make is that the formulation presented in the previous section (*i.e.* binary logistic regression) is exclusively for the case where the number of possible outcomes for the categorical target variable is exactly two. There is no immediate generalization of the binary case which would work for the multinomial case.

Before we formally generalize the logistic regression to the multinomial case, we introduce an important function, namely the **softmax function**. The softmax function appears very often in machine learning (and in particular in deep learning), and you need to have a solid understanding of this important function. The softmax function is defined by

$$\begin{aligned} &\text{softmax} : \mathbb{R}^n \to (0,1)^n \\ &\text{softmax}(z_1, z_2, \cdots, z_n) = \left(\frac{e^{z_1}}{\sum_{i=1}^n e^{z_i}}, \frac{e^{z_2}}{\sum_{i=1}^n e^{z_i}}, \cdots, \frac{e^{z_n}}{\sum_{i=1}^n e^{z_i}} \right). \end{aligned} \tag{16}$$

Note that the input of the softmax function is an n-tuple and its output is another n-tuple. The output of the softmax function possesses two crucial properties:

- Note that every element of the output n-tuple $\dfrac{e^{z_j}}{\sum_{i=1}^n e^{z_i}} = \dfrac{e^{z_j}}{e^{z_1} + \cdots + e^{z_n}}$ belongs to the interval $(0,1)$. Therefore, every element of the output n-tuple can be interpreted as a **probability**!
- The sum of the n elements of the output n-tuple is 1: $\displaystyle\sum_{j=1}^{n} \frac{e^{z_j}}{\sum_{i=1}^n e^{z_i}} = \frac{e^{z_1} + e^{z_2} + \cdots + e^{z_n}}{e^{z_1} + e^{z_2} + \cdots + e^{z_n}} = 1.$
  This shows that the output of the softmax function is a **discrete probability distribution**!

The essence of the above argument can be summarized as follows: Suppose we have n entities, and we assign score to these entities. By passing the scores to the softmax function, we convert the scores into a discrete probability distribution. The individual entries of the output of the softmax function correspond to the probabilities of occurrences. The fact that the probabilities add up to 1 indicates that one of the cases will certainly take place.

Let us define the softmax function in python using numpy, and see how the function works in practice.

[2]:
```
def softmax(x):                    # Define softmax function
    e = np.exp(x)
    return e/e.sum()

examples = [[100, 100], [7, 5], [1, 2, 0], [4, 2, -100], [1, 2, 3, 4, 5], [-1,␣
 ↪2, 0, 1, -1, 3]]

for x in examples:
    print('Softmax(%s)=%s\n' %(x, softmax(x)))
```

```
Softmax([100, 100])=[0.5 0.5]
```

```
Softmax([7, 5])=[0.88079708 0.11920292]

Softmax([1, 2, 0])=[0.24472847 0.66524096 0.09003057]

Softmax([4, 2, -100])=[8.80797078e-01 1.19202922e-01 6.00136094e-46]

Softmax([1, 2, 3, 4, 5])=[0.01165623 0.03168492 0.08612854 0.23412166
0.63640865]

Softmax([-1, 2, 0, 1, -1, 3])=[0.01152193 0.23142412 0.03131985 0.08513618
0.01152193 0.62907599]
```

Exercise: Calculate the softmax of $(3,5,6,8)$, $(3.5,5.5,6.5,8.5)$, $(-2,0,1,3)$, $(5,7,8,10)$. What do you conclude from this exercise?

[3]:
```
four_tuples = [[3, 5, 6, 8], [3.5, 5.5, 6.5, 8.5], [-2, 0, 1, 3], [5, 7, 8, 10]]

for x in four_tuples:   # Calculating the softmax for above 4-tuples
    print('Softmax(%s)=%s\n' %(x, softmax(x)))
```

```
Softmax([3, 5, 6, 8])=[0.0056533  0.04177257 0.11354962 0.83902451]

Softmax([3.5, 5.5, 6.5, 8.5])=[0.0056533  0.04177257 0.11354962 0.83902451]

Softmax([-2, 0, 1, 3])=[0.0056533  0.04177257 0.11354962 0.83902451]

Softmax([5, 7, 8, 10])=[0.0056533  0.04177257 0.11354962 0.83902451]
```

The above exercise indicates that the softmax function enjoys an important symmetry! If we shift all elements of the input n-tuple by the same constant, the softmax function stays constant:

$$\boxed{\text{softmax}(z_1+c, z_2+c, \cdots, z_n+c) = \text{softmax}(z_1, z_2, \cdots, z_n)}\,. \tag{17}$$

Now that we know the softmax function and its basic properties, we can formulate logistic regression for the multinomial classification. Suppose we have a classification problem with K classes. This means that there are K possible values that the categorical target variable can assume. In the case of binary classification, we modeled the probability function with the sigmoid function. In the multinomial case, we model the probabilities of for different outcomes with the softmax function. We need to specify the probability functions $P(y^{(i)}=k|\vec{x}^{(i)})$ (*i.e.* the probability that the target y is observed in class k in the i-th observation, given that the observed features are $\vec{x}^{(i)}$). We can specify all probabilities at once by

$$\Big(P(y^{(i)}=1|\vec{x}^{(i)}), P(y^{(i)}=2|\vec{x}^{(i)}), \cdots, P(y^{(i)}=K|\vec{x}^{(i)})\Big) = \text{softmax}(z_1^{(i)}, z_2^{(i)}, \cdots, z_K^{(i)})\,, \tag{18}$$

where $z_j^{(i)} = \omega_{0,j} + \vec{\omega}_j \cdot \vec{x}^{(i)} = \omega_{0,j} + \omega_{1,j}x_1^{(i)} + \omega_{2,j}x_2^{(i)} + \cdots + \omega_{d,j}x_d^{(i)}$ for $j = 1, 2, \cdots, K$.

Important Note: In the case of multinomial classification, a *new set of weights* ($\omega_{0,j}$ and $\vec{\omega}_j$ for $j = 1, 2, \cdots, K$) is introduced for *each class*. The total number of parameters of the multinomial logistic regression model is $K(d+1)$, with K being the number of classes and d the number of features. The parameters of the model can be accommodated in a $(d+1) \times K$ matrix:

$$\omega = \begin{pmatrix} \omega_{0,1} & \omega_{0,2} & \cdots & \omega_{0,K} \\ \omega_{1,1} & \omega_{1,2} & \cdots & \omega_{1,K} \\ \vdots & \vdots & \ddots & \vdots \\ \omega_{d,1} & \omega_{d,2} & \cdots & \omega_{d,K} \end{pmatrix} . \tag{19}$$

Each column of (19) casts the weights associated with a class of the target variable. The properties of the softmax function guarantees that through (18), we get a discrete probability distribution with K classes. As usual, in order to find the optimal values of the parameters (weights ω in (19)), we need to minimize an appropriate cost function. The cost function for the multinomial case is a generalization of the cost function of the binary logistic regression. The multinomial logistic regression cost function is defined by

$$J(\omega) = -\sum_{i=1}^{n}\sum_{k=1}^{K} \log\left(P(y^{(i)} = k|\vec{x}^{(i)})\right) = -\sum_{i=1}^{n}\sum_{k=1}^{K} \log\left[\frac{e^{z_k^{(i)}}}{\sum_{j=1}^{K} e^{z_j^{(i)}}}\right] . \tag{20}$$

Similar to the case of binary classification, the optimization equations (*i.e.* derivatives of $J(\omega)$ with respect to the weights ω being set to zero) cannot be solved analytically. scikit-learn library uses an iterative gradient descent approach to solve the optimization equations numerically.

We close the discussion on the multinomial logistic regression by two side remarks.

Side Remark: Due to property (17), the multinomial logistic model is *overparameterized*! We can use the symmetry property (17) to reduce the number of parameters to $(d+1)(K-1)$. In other words, property (17) can be used to get rid of one of the columns of (19).

Side Remark: It can be shown that the multinomial logistic regression with two classes (*i.e.* $K = 2$) is equivalent to the binary logistic regression when one uses property (17).

1.3 Metrics to Evaluate Classifiers

In the previous section, we introduced the very first classifier, namely the logistic regression. It is now time to illustrate how we should evaluate classifiers, and measure how good they perform. To achieve this goal, we need to introduce appropriate criteria by which classifiers can be assessed. Recall that when we introduced regression models in the previous sessions, we saw how one was supposed to assess regressors. In this section, we would like to do the same thing for classifiers. Note that the same metrics (*e.g.* RSE, R^2-score, F-statistic, Adjusted R^2-score, etc.) introduced for assessing regressors *cannot be applied* to classifiers, and we first need to come up with the right metrics that can be applied to classifiers.

In order to make comparisons between classifiers, we first introduce two boring classifiers. These two classifiers help you understand why your constructed classifier works the way it does. In every real world classification problem, you need to make sure that your classifier performs better

than these two classifiers. Moreover, to interpret your model's performance, it is important to establish by how much your model beats both of these boring classifiers.

- **Sharp Classifier:** A sharp classifier labels *all observations* with one class regardless of the values the features assume. Suppose you are working on a binary classification problem with the goal to predict the gender based on the height. A sharp classifier labels *all observations* with one class (*i.e.* male or female) regardless of the height.
- **Monkey Classifier:** A monkey classifier labels each observation randomly (based on pure chance). In other words, if there are K classes for the target, the probability of choosing each of the K classes for the monkey classifier is $\frac{1}{K}$, regardless of the values the features assume. For instance, if we are dealing with a binary classification problem, the monkey classifier picks either the Success ($y = 1$) or the Failure ($y = 0$) with a 50-50 chance for each instance (*i.e.* monkey classifier tosses a fair coin for each observation), regardless of the values the features assume in each instance.

Both the sharp and the monkey classifiers are very boring, and in most of the real life problems, they perform poorly. However, when we construct a genuine classifier, it is important to figure out *by how much* our classifier does a better job than the two boring classifiers defined above!

1.3.1 Metrics to Evaluate Binary Classifiers

Now, we are ready to define the metrics by which we assess classifiers. We first introduce these metrics in the context of binary classifications, and then generalize them to the multinomial case. To be consistent with the machine learning literature, instead of labeling the two outcomes of the categorical target variable by Success and Failure, we label them by **positive** and **negative**. In this labeling the negative label is used for the bigger class, and the positive label is used for the smaller class. If the sizes of the two classes are equal, you can label either of the them with the positive and the other one with the negative label. In assessing different classifiers, it is important to make a distinction between the two following types of qualitatively different classification problems:

- **Balanced Case:** In a balanced binary classification problem, the size of the positive class is close to the size of the negative class (Note that still the size of the negative class is, by definition, greater than the size of the positive class, but in the balanced case, not much bigger). In other words, the sizes of the two classes are comparable. An example of a balanced classification is to predict the gender of fair sample of people based on their height. The size of the positive class (say males) are more or less the same as the size of the negative class (say females).
- **Unbalanced Case:** In an unbalanced binary classification problem, the size of the negative class is much bigger (orders of magnitude bigger) than the size of the positive class. A prominent example of an unbalanced binary classification is to predict whether individuals chosen from a fair sample of participants are diagnosed with cancer. In this example, the size of the positive class (*i.e.* cancer positive) is much much smaller than the size of the negative class (*i.e.* cancer negative).

In each binary classification problem, in order to understand the behavior of your classifier, you should ask yourself which type of classification (balanced or unbalanced) you are dealing with.

With the terminology introduced so far, we can imagine four possible cases to take place for each instance in a binary classification problem:

- **True Positive:** A classifier labels a positive instance as positive, resulting in a win for the classifier.
- **True Negative:** A classifier labels a negative item as negative, resulting in a win for the classifier.
- **False Positive:** A classifier mistakenly labels a negative item as a positive, resulting in a **Type I** classification error.
- **False Negative:** A classifier mistakenly labels a positive item as a negative, resulting in a **Type II** classification error.

We can conveniently represent the sizes of the above four distinct cases through a matrix. This matrix is called the **Confusion Matrix**, as is defined by

$$CM = \begin{pmatrix} TP & FN \\ FP & TN \end{pmatrix} , \tag{21}$$

where TP, FN, FP, and TN represent the total number of true positives, false negatives, false positives, and true negatives, respectively.

Accuracy Statistic The first metric defined to measure the performance of a classifier is the **accuracy**. Accuracy is defined as the ratio of the number of correct predictions over total predictions:

$$accuracy = \frac{TP + TN}{TP + TN + FN + FP} . \tag{22}$$

Multiplying the fraction in (22) by 100, accuracy is usually expressed in percentages. Also note that accuracy is a metric that is defined for the classification altogether (*i.e.* It is not defined for classes individually). The rest of the metrics that we will study are defined for each class of the classification problem. Accuracy is a metric that is relatively easy to explain, and it is worth providing in any classification evaluation.

Now, let us contemplate about the accuracy of the sharp and the monkey classifiers in a balanced binary classification problem. In a balanced binary classification, the monkey classifier is expected to achieve an accuracy around 50% by (pure) random guessing. The sharp classifier would achieve the same accuracy because for the sharp all predictions are positive (they could be all negative as well), and hence: $TN = FN = 0$, and $TP = FP = \frac{n}{2}$ with n being the total number of instances. This implies that a poor classifier (like the monkey or the sharp) can be 50% accurate in a balanced binary classification problem, and hence, a genuine classifier must considerably be more accurate than 50%!

Limitation of Accuracy Alone: Accuracy *alone* has, however, limitations as an evaluation metric. To show the limitations of accuracy (as the only evaluation metric), we consider an unbalanced binary classification problem. Consider the cancer classification problem from a fair sample. In a fair sample of people, the majority of the participants are healthy. Let us assume that the chance of finding people with cancer in 5%. Now suppose a sharp classifier labels all participants as

healthy (*i.e.* the sharp classifier puts everybody in the negative class). What would be the accuracy of this sharp classifier? Well, there is no positive class predictions, hence $TP = FP = 0$. Thus, $accuracy = \frac{TN}{TN+FN} = \frac{0.95\,n}{0.95\,n+0.05\,n} = 0.95$. Therefore, the accuracy of the sharp classifier is 95%! We know that a sharp classifier is a very weak classifier. Nonetheless, it achieves a remarkably high accuracy! This shows that accuracy cannot be trusted as the only evaluation metric. In particular, in dealing with unbalanced classification problems, a high accuracy alone does not mean anything!

In order to offer a wider range of metrics to evaluate classifiers, we define three more metrics, namely *precision*, *recall*, and *F1-score*. Note that unlike the accuracy, these three metrics are defined for each class of the target variable!

Precision Statistic Precisions for the positive and negative classes in a binary classification problem are defined as follows:

$$\begin{aligned} precision[P] &= \frac{TP}{TP+FP}\,, \\ precision[N] &= \frac{TN}{TN+FN}\,. \end{aligned} \tag{23}$$

If we now go back to the unbalanced classification example (*i.e.* predicting cancer patients) at the end of the previous section, we realize that since for the sharp classifier $TP = FP = 0$, it is not even possible to define $precision[P]$! This is a bad sign, as a strong classifier should produce precision close to 1. Although one gets a high accuracy in the cancer classification problem, it should become evident that the sharp classifier has a problem in producing a viable result for the precision of the positive class (even though the $precision[N]$ is still high), and this is a sign of trouble! It would be impossible for a sharp classifier to achieve high precision on both classes ($[P]$ and $[N]$) in any real classification problem.

Recall Statistic Recalls for the positive and negative classes in a binary classification problem are defined as follows:

$$\begin{aligned} recall[P] &= \frac{TP}{TP+FN}\,, \\ recall[N] &= \frac{TN}{TN+FP}\,. \end{aligned} \tag{24}$$

Note: The recall of the positive class ($recall[P] = \frac{TP}{TP+FN}$) is sometimes referred to as **sensitivity** in the machine learning literature.

It is straightforward to see that recall is also able to detect the deficiency of the sharp classifier in the cancer classification problem. In that example, the recall of the negative class is high ($recall[N] = 1$ as $FP = 0$), but the recall of the positive class is 0 ($recall[P] = \frac{0}{0+0.05\,n} = 0$)! This means that the sharp classifier performs very poor on the positive class (a fact we already knew!).

Question: Why do we need both precision and recall? Can't we live with one of them?

Answer: Sometimes we might be willing to tolerate false positives more than false negatives. For instance, in the context of the cancer classification problem, a false positive case is a participant who does not actually have cancer, but has mistakenly been labeled as a cancer patient. On the other hand, a false negative case is a cancer patient who has been labeled as healthy. Due to the severe consequences, it is evident that we can tolerate false positive cases, but our tolerance for false negative is very low! From (24), it is straightforward to see that a high $recall[P]$ (close to 1) guarantees that your classifier has a very few false negatives!

Exercise: Why can't we use $precision[N]$ in the same manner to make sure that we have very few false negatives?

Important Remark: In training classifiers, increasing precision of a class decreases the recall of the same class, and vice versa. This phenomenon is referred to as the **precision/recall trade-off**.

***F*1-Score** $F1$-score is defined as the *harmonic mean* of the precision and recall for each class. First, let us quickly explain what a harmonic mean is. There are three types of mean defined in statistics: the *arithmetic mean*, the *geometric mean*, and the *harmonic mean*. Arithmetic mean is the most famous type of mean that you have worked with. Suppose $x_1, x_2, \cdots, x_n$ are n real numbers. The harmonic mean H of $x_1, x_2, \cdots, x_n$ is given by

$$H = \frac{n}{\frac{1}{x_1} + \frac{1}{x_2} + \cdots + \frac{1}{x_n}} = \frac{n}{\sum_{i=1}^{n} \frac{1}{x_i}} \,. \tag{25}$$

Now, the $F1$-scores for the positive and negative classes in a binary classification are defined by

$$\begin{aligned} F1[P] &= \frac{2\, precision[P]\, recall[P]}{precision[P] + recall[P]} \,, \\ F1[N] &= \frac{2\, precision[N]\, recall[N]}{precision[N] + recall[N]} \,. \end{aligned} \tag{26}$$

If one is looking for a single metric to make a good judgement about the performance of a classifier, $F1$-score would be that metric. $F1$-score is a very tough measure to beat! That is essentially because the harmonic mean is always less than or equal to the arithmetic mean (Note that $F1$-score is the harmonic mean of the precision and recall for each class), and the smaller number has a larger effect. Achieving a *high $F1$-score requires both high precision and high recall!* None of the poor classifiers (*e.g.* the monkey or the sharp) can produce a decent $F1$-score despite high accuracy and recall.

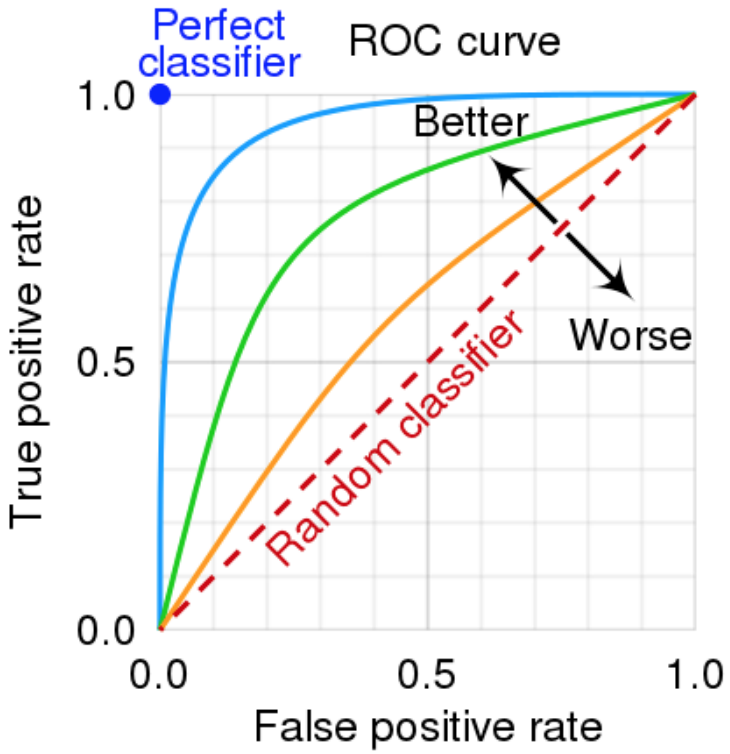
Lessons to Take Away:

1. Accuracy is a misleading metric in unbalanced classification problems!
2. High precision is very hard to achieve in unbalanced classification problems!
3. $F1$-score does the best job of any single metric, but all four of them work together to evaluate the performance of a classifier!

Receiver-Operator Characteristic (ROC) Receiver-Operator Characteristic (ROC) is a metric to solely evaluate *binary classifiers* (ROC is not applicable to multinomial classifiers). ROC is a graphic method of evaluating binary classifiers in which one plots the true positive rate versus the false positive rate at various thresholds. The method was originally developed for operators of military radar receivers starting in 1941, which led to its name. The **true positive rate** (TPR) and the **false positive rate** (FPR) for a binary classifier are defined by

$$\begin{aligned} TPR &= \frac{TP}{TP+FN}\,, \\ FPR &= \frac{FP}{FP+TN}\,. \end{aligned} \tag{27}$$

Note that TPR is the same as the *sensitivity* (or equivalently $recall[P]$), and FPR is equal to $FPR = 1 - specificity$, where **specificity** is defined as $specificity = \frac{TN}{TN+FP}$. Since TPR and FPR are both rates, and hence the plot only concerns the square $[0,1] \times [0,1]$. The diagonal (*i.e.* the line connecting the origin $(0,0)$ to the point $(1,1)$) in this plot describes the performance of the monkey classifier. To evaluate your classifier, you need to calculate its corresponding TPR and FPR. Then plot the point (TPR, FPR) in the ROC plot. If the point stands above the diagonal, the performance of your classifier is better than the monkey classifier. On the other hand, if the point stands below of the diagonal, your classifier performs worse than a monkey!



In order to measure by how much a classifier performs better than a monkey classifier, we connect the point (TPR, FPR) to the origin $(0,0)$ and to $(1,1)$. The Area Under the Curve (AUC) is a metric that shows by how much a given classifier performs better than the monkey classifier. Note that AUC for the monkey classifier is $AUC = \frac{1}{2}$ (*i.e.* the area of a half square). For a **perfect classifier**, the (TPR, FPR) point is $(0,1)$, and hence $AUC = 1$. It is then evident that the AUC of a classifier

which performs better than the monkey, but is not perfect, is $\frac{1}{2} < AUC < 1$.

Note: One can easily show that AUC (*i.e.* the area under the ROC curve) is equal to the arithmetic mean of sensitivity and specificity (*i.e.* $AUC = \frac{sensitivity + specificity}{2}$).

Note: scikit-learn library can easily plot the ROC curve of any binary classifier, and automatically shows its corresponding AUC in the ROC plot.

1.3.2 Metrics to Evaluate Multinomial Classifiers

We now generalize the definitions for the evaluation metrics offered for the binary classification in the previous section to the multinomial case. Suppose, we are dealing with a classification problem with K classes. Let $C[i,j]$ denote the number of instances of class i that received label j by the classifier. In other words, i (the first index) in $C[i,j]$ denotes the true (actual) label and j (the second index) denotes the predicted label. We can then define a $K \times K$ **confusion matrix** as follows:

$$CM = \begin{pmatrix} C[1,1] & C[1,2] & \cdots & C[1,K] \\ C[2,1] & C[2,2] & \cdots & C[2,K] \\ \vdots & \vdots & \ddots & \vdots \\ C[K,1] & C[K,2] & \cdots & C[K,K] \end{pmatrix} . \tag{28}$$

Note that the diagonal elements ($C[i,i]$) of the confusion matrix (28) represent the number of true predictions for each class, and the off diagonal elements ($C[i,j]$ with $i \neq j$) of (28) represent the number false predictions of different pairs of classes. Equation (28) is the generalization of the confusion matrix (21) in the binary case.

The accuracy in the multinomial classification is then defined by

$$accuracy = \frac{\sum_{i=1}^{K} C[i,i]}{\sum_{j=1}^{K}\sum_{k=1}^{K} C[j,k]} . \tag{29}$$

The above equation is the generalization of (22) in the binary case. The precision for each class $[i]$ of the classification is then defined by

$$precision[i] = \frac{C[i,i]}{\sum_{j=1}^{K} C[j,i]} . \tag{30}$$

In the denominator of (30), we consider all instances with predicted label $[i]$. This formula, (30), is consistent with the formula (23) in the binary case. Similarly, the recall for each class $[i]$ is defined by

$$recall[i] = \frac{C[i,i]}{\sum_{j=1}^{K} C[i,j]} . \tag{31}$$

In the denominator of (31), we consider all instances with actual label $[i]$. The above formula, (31), is consistent with the definition (24) in the binary case. Finally, the $F1$-score for each class $[i]$ is calculated as the harmonic mean of the precision and recall of the same class:

$$F1[i] = \frac{2\, precision[i]\, recall[i]}{precision[i] + recall[i]} . \tag{32}$$

The above formula is the same as (26).

Note: scikit-learn library calculates the confusion matrix and all the above mentioned metrics for a classifier and represents the summary in a nice manner.

2 Coding for Logistic Regression

In here, we consider two examples of classification problems to be solved by the logistic regression. The first example is a *binary classification problem* and the second will be a *multinomial classification problem*.

2.1 A Binary Example

In our first example, the major weather indicators of Seattle's weather have been recorded for more than 25000 consecutive days from 1948 to 2017. In this dataset, the minimum and the maximum temperatures of each day of the year for the period 1948 to 2017 have been recorded. Moreover, the rain condition for each day has been recorded as a True/False boolean variable. In a separate column, the precipitation rates for rainy days have been recorded as well. In this example, we employ logistic regression to build a binary classifier for predicting the weather condition of Seattle. The two classes in example are `raniy` and `not rainy`.

[4]:
```
# Importing basic libraries

import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
%matplotlib inline
import seaborn as sns
import mlxtend

df=pd.read_csv('../Week-4/SeattleWeather.csv')      # Reading the csv source file␣
 ↪as a dataframe
df.shape
```

[4]:
```
(25551, 5)
```

[5]:
```
df.sample(5)     # A sample of 5 random rows of 'df'
```

[5]:

	DATE	PRCP	TMAX	TMIN	RAIN
16314	1992-08-31	0.00	76	58	False
21561	2007-01-12	0.00	31	19	False
11221	1978-09-21	0.18	62	52	True
22671	2010-01-26	0.00	48	38	False
11103	1978-05-26	0.06	58	50	True

[6]:
```
# Defining a target variable that casts the rain condition (0 for not rainy, and␣
 ↪1 for rainy)

df['Target'] = df.RAIN.map(lambda x: 1 if x==True else 0)
df.sample(5)
```

[6]:

	DATE	PRCP	TMAX	TMIN	RAIN	Target
23647	2012-09-28	0.00	77	54	False	0
22398	2009-04-28	0.12	54	45	True	1
4512	1960-05-09	0.00	76	47	False	0
24285	2014-06-28	0.09	68	56	True	1
20898	2005-03-20	0.10	55	40	True	1

[7]:
```
#Defining 'TMIN' and 'TMAX' as features and the target as numpy arrays

X = df[['TMIN','TMAX']].to_numpy()
y = df.Target.to_numpy()
```

[8]:
```
# Dividing the features and the target variable into the train and test subsets

from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3,␣
 ↪random_state=5, stratify=y)
```

[9]:
```
# Checking the distribution of the train and test categories for the target␣
 ↪variable

print(np.bincount(y))
print(np.bincount(y_train))
print(np.bincount(y_test))
```

```
[14651 10900]
[10255  7630]
[4396 3270]
```

[10]:
```
from sklearn.linear_model import LogisticRegression    # Importing Logistic␣
 ↪Regression from sklearn

logreg = LogisticRegression(penalty = 'none', max_iter = 10000) # Instantiating␣
 ↪logistic regression

logreg.fit(X_train, y_train)  # Fitting the train data to 'logreg'
```

[10]:
```
LogisticRegression(max_iter=10000, penalty='none')
```

Note that you can *regularize* the logistic regression if it is necessary. In case a regularization is necessary, you can choose either $L1$ or $L2$ penalty (penalty='l1' or penalty='l2'), and then you need to specify the strength of the penalty term by C=0.7 for instance.

Also note that since scikit-learn uses a gradient descent approach in solving the optimization problem, you have the option to choose the maximum number of iterations through max_iter= option.

Question: How do we figure out if a penalty term (regularization) is necessary for the logistic regression classifier?

[11]:
```
# Predicted class for the test subset through logistic regression

y_train_pred_prob = logreg.predict_proba(X_train)  # Calculating probabilities␣
 ↪of the two classes in train subset
y_test_pred_prob = logreg.predict_proba(X_test)    # Calculating probabilities␣
 ↪of the two classes in test subset

# Presenting the probabilities for the 1st 10 instances in train subset

print('Train predictions for probabilities:\n\n', y_train_pred_prob[:10],'\n')

# Presenting the probabilities for the 1st 10 instances in test subset

print('Test predictions for probabilities:\n\n', y_test_pred_prob[:10])
```

```
Train predictions for probabilities:

 [[0.35200246 0.64799754]
 [0.27913347 0.72086653]
 [0.37547161 0.62452839]
 [0.57136744 0.42863256]
 [0.28272534 0.71727466]
 [0.56117778 0.43882222]
 [0.94079684 0.05920316]
 [0.70590084 0.29409916]
 [0.58293863 0.41706137]
 [0.57860941 0.42139059]]
```

```
Test predictions for probabilities:

 [[0.18758714 0.81241286]
 [0.62558783 0.37441217]
 [0.89430244 0.10569756]
 [0.45018241 0.54981759]
 [0.90953918 0.09046082]
 [0.32937617 0.67062383]
 [0.4022255  0.5977745 ]
 [0.4022255  0.5977745 ]
 [0.51537775 0.48462225]
 [0.64752248 0.35247752]]
```

[12]:
```
y_train_pred = y_train_pred_prob.argmax(axis=1)  # Finding the predicted class␣
 ↪for train predictions
y_test_pred = y_test_pred_prob.argmax(axis=1)  # Finding the predicted class for␣
 ↪test predictions

# Presenting the predicted classes for the 1st 10 instances of train

print('Predicted class for train: ', y_train_pred[:10],'\n')

# Presenting the predicted classes for the 1st 10 instances of test

print('Predicted class for train: ', y_test_pred[:10])
```

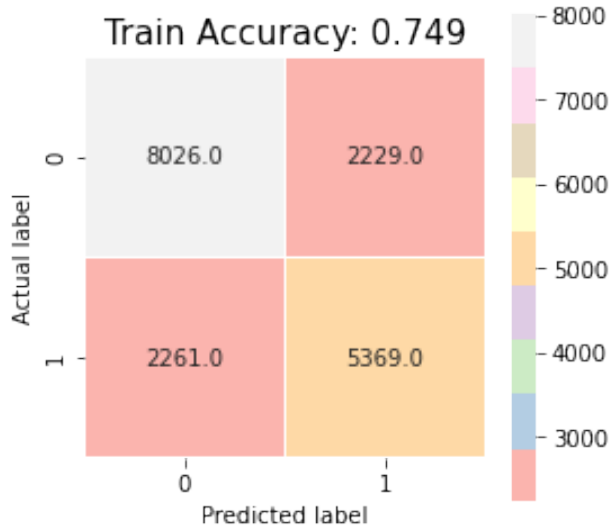
```
Predicted class for train:  [1 1 1 0 1 0 0 0 0 0]

Predicted class for train:  [1 0 0 1 0 1 1 1 0 0]
```

[13]:
```
# Calculating the train and test accuracy scores of the model

from sklearn import metrics    # Importing 'metrics' from sklearn

train_score = metrics.accuracy_score(y_train, y_train_pred) # train accuracy

test_score = metrics.accuracy_score(y_test, y_test_pred)    # test accuracy

print('Train accuracy score of the model is ', round(train_score, 5),'\n')
print('Test accuracy score of the model is ', round(test_score, 5))
```

```
Train accuracy score of the model is  0.74895

Test accuracy score of the model is  0.7575
```

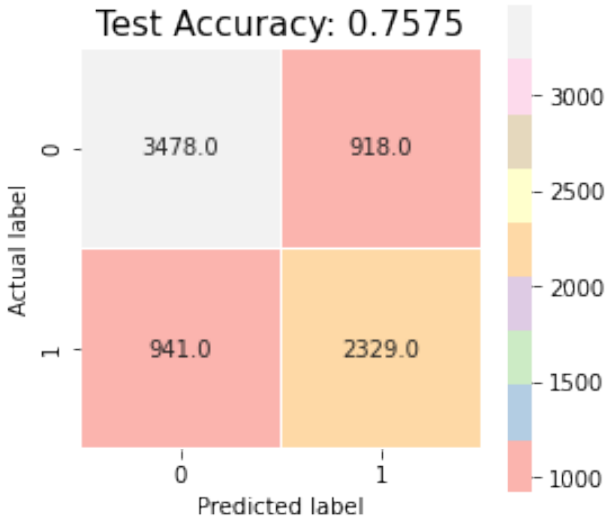
[14]:
```
# Computing the confusion matrix for the train subset
```

```
from sklearn.metrics import confusion_matrix, classification_report

train_c_matrix = confusion_matrix(y_train, y_train_pred)  # Compute the train␣
 ↪confusion matrix

plt.figure(figsize=(4,4))
sns.heatmap(train_c_matrix, annot=True, fmt=".1f", linewidths=.5, square = True,␣
 ↪cmap = 'Pastel1');
plt.ylabel('Actual label');
plt.xlabel('Predicted label');
all_sample_title = 'Train Accuracy: {0}'.format(round(train_score, 4))
plt.title(all_sample_title, size = 15);
plt.show()

# Print the train classification report

print('Classification Report for the Train Subset: \n\n',␣
 ↪classification_report(y_train, y_train_pred))
```




```
Classification Report for the Train Subset:

              precision    recall  f1-score   support

           0       0.78      0.78      0.78     10255
           1       0.71      0.70      0.71      7630

    accuracy                           0.75     17885
   macro avg       0.74      0.74      0.74     17885
weighted avg       0.75      0.75      0.75     17885
```

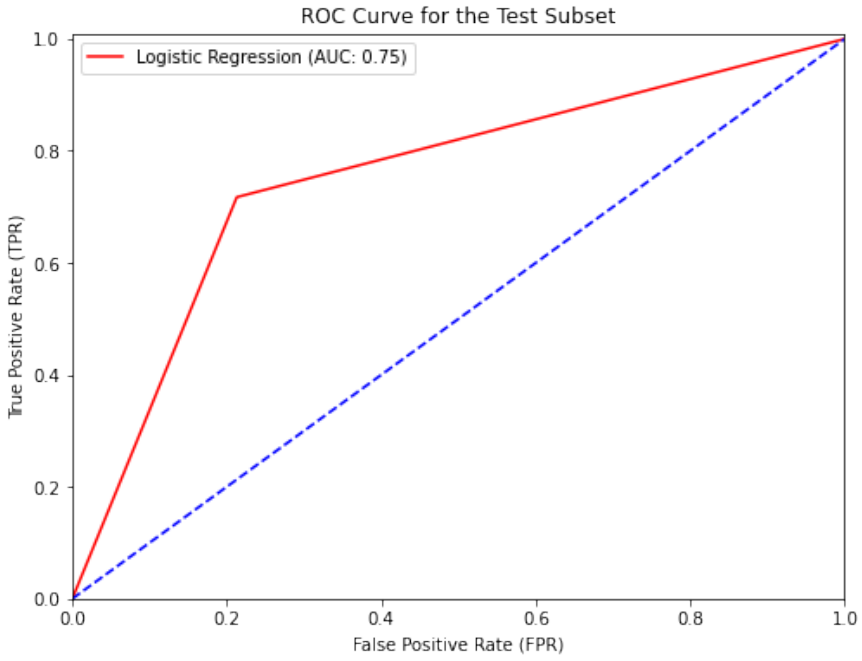
[15]:
```
# Computing the confusion matrix for the test subset

test_c_matrix = confusion_matrix(y_test, y_test_pred)  # Compute the test␣
 ↪confusion matrix

plt.figure(figsize=(4,4))
sns.heatmap(test_c_matrix, annot=True, fmt=".1f", linewidths=.5, square = True,␣
 ↪cmap = 'Pastel1');
plt.ylabel('Actual label');
plt.xlabel('Predicted label');
all_sample_title = 'Test Accuracy: {0}'.format(round(test_score, 4))
plt.title(all_sample_title, size = 15);
plt.show()

# Print the train classification report

print('Classification Report for the Test Subset: \n\n',␣
 ↪classification_report(y_test, y_test_pred))
```



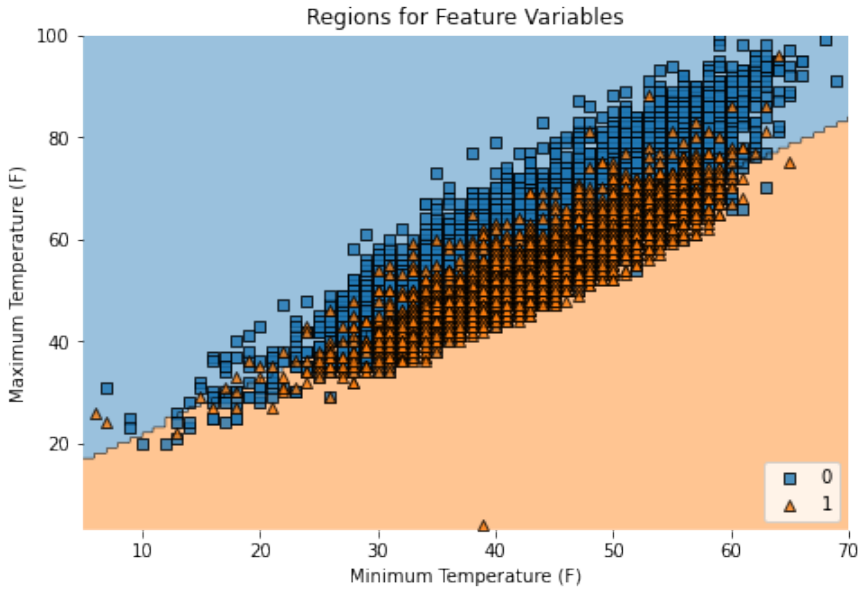

```
Classification Report for the Test Subset:

              precision    recall  f1-score   support

           0       0.79      0.79      0.79      4396
           1       0.72      0.71      0.71      3270

    accuracy                           0.76      7666
```

```
   macro avg       0.75      0.75      0.75      7666
weighted avg       0.76      0.76      0.76      7666
```

[16]:
```
# Plotting the ROC curve for the test subset

from sklearn.metrics import roc_curve, auc   # Importing 'roc_curve' and 'auc'␣
 ↪from sklearn

fpr, tpr, thresholds = roc_curve(y_test_pred, y_test)  # Computing ROC for the␣
 ↪test subset
auc(fpr, tpr)                                          # Computing AUC for the␣
 ↪test subset

plt.figure(figsize=(8, 6))
plt.plot(fpr, tpr, color='red', label='Logistic Regression (AUC: %.2f)'
% auc(fpr, tpr))
plt.plot([0, 1], [0, 1], color='blue', linestyle='--')
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.01])
plt.title('ROC Curve for the Test Subset')
plt.xlabel('False Positive Rate (FPR)')
plt.ylabel('True Positive Rate (TPR)')
plt.legend()
plt.show()
```



[17]:
```
# Plotting the decision regions

from mlxtend.plotting import plot_decision_regions

mlxtend.plotting.plot_decision_regions(X=X_test, y=y_test, clf=logreg) #␣
 ↪Plotting the decision region

plt.title('Regions for Feature Variables')
plt.xlabel('Minimum Temperature (F)')
plt.ylabel('Maximum Temperature (F)')
plt.legend(loc='lower right')
plt.tight_layout()
plt.gcf().set_size_inches(7, 5)
plt.show()

## 0: No Rain
## 1: Rain
```



Question: Based on what you see in the decision region plot above, can you explain why the constructed binary classifier cannot achieve an accuracy better than 75%?

2.2 A Multinomial Example

The **MNIST database** (for further information visit http://yann.lecun.com/exdb/mnist/) consists of 70000 images of handwritten digits. The digits have been size-normalized and centered in a fixed-size image. Each image has been fitted into a $28 \times 28 = 784$ pixel box. scikit-learn library is automatically equipped with the MNIST dataset through fetch_openml. In this classification prob-

lem, we treat the 784 pixel inputs of each image as features and the target variable is the actual digit the image represents, with 10 distinct classes (0, 1, 2, 3, 4, 5, 6, 7, 8, 9).

To implement this classification problem, we put a cap on the number of images we take from MNIST dataset because including all 70000 images requires a long computation time.

[18]:
```
from sklearn.datasets import fetch_openml    # Importing 'fetch_openml' from␣
 ↪sklearn

# Downloading features and target from MNIST dataset

X, y = fetch_openml('mnist_784', version=1, return_X_y=True, as_frame=False)

X.shape  # Shape of features as numpy array
```

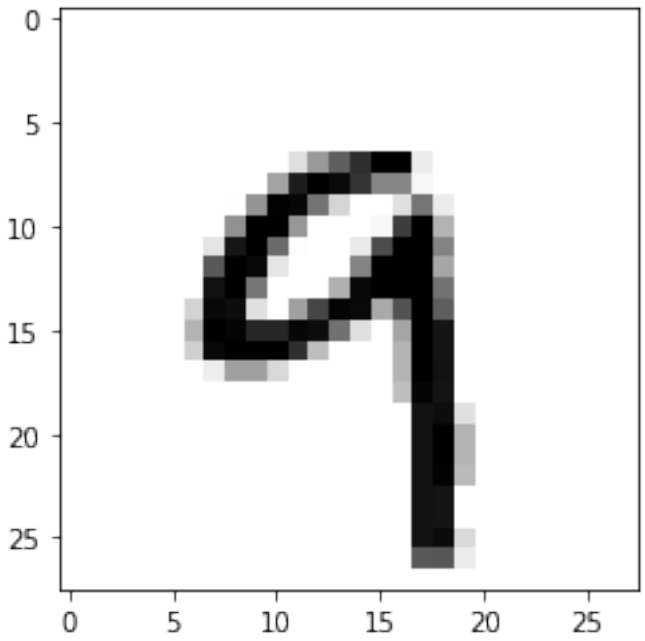
[18]:
```
(70000, 784)
```

[19]:
```
# Convert classes from strings to integers

y = np.array([int(a) for a in y])

y
```

[19]:
```
array([5, 0, 4, ..., 4, 5, 6])
```

[20]:
```
# Constructing the images of MNIST

def digit_image(sample):
    image = X[sample].reshape(28, 28)  # Reshape the feature 'X' as 28*28 arrays
    fig = plt.figure
    plt.imshow(image, cmap='gray_r')
    plt.show()
    digit = 'Digit Class is: %s' %(y[sample])
    return print(digit)
```

[21]:
```
# Presenting the image of the sample

digit_image(932)
```




```
Digit Class is: 9
```

[22]:
```
# Let's see how digits are distributed in the first 10000 MNIST images

count = 0
for j in range(10):
    z = [1 if a==j else 0 for a in y[:10000]]
    count += sum(z)
    print('The number of digit %d in the subset is: %d' %(j, sum(z)))
print('Total number of instances in the subset: ', count)
```

```
The number of digit 0 in the subset is: 1001
The number of digit 1 in the subset is: 1127
The number of digit 2 in the subset is: 991
The number of digit 3 in the subset is: 1032
The number of digit 4 in the subset is: 980
The number of digit 5 in the subset is: 863
The number of digit 6 in the subset is: 1014
The number of digit 7 in the subset is: 1070
The number of digit 8 in the subset is: 944
The number of digit 9 in the subset is: 978
Total number of instances in the subset:  10000
```

We realize that each class has roughly about 1000 images. This indicates that we are dealing with a fairly *balanced classification* problem.

[23]:
```
# Dividing the features and the target variable into the train and test subsets

cap = 10000

XX = X[:cap]
yy = y[:cap]

from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(XX, yy, test_size=0.3,␣
 ↪random_state=5)
```

[24]:
```
from sklearn.linear_model import LogisticRegression   # Importing Logistic␣
 ↪Regression from sklearn

logreg = LogisticRegression(penalty = 'none', max_iter = 10000) # Instantiating␣
 ↪logistic regression

import timeit

start = timeit.default_timer()
logreg.fit(X_train, y_train)  # Fitting the train data to 'logreg'
stop = timeit.default_timer()

print('Training time: %s sec' %(stop-start))
```

```
Training time: 2.860097321999998 sec
```

[25]:
```
# Predicted class for the test subset through logistic regression

y_train_pred_prob = logreg.predict_proba(X_train)  # Calculating probabilities␣
 ↪of the two classes in train subset
y_test_pred_prob = logreg.predict_proba(X_test)    # Calculating probabilities␣
 ↪of the two classes in test subset

# Presenting the probabilities for the 1st 10 instances in train subset

print('Train predictions for probabilities:\n\n', y_train_pred_prob[:3],'\n')

# Presenting the probabilities for the 1st 10 instances in test subset

print('Test predictions for probabilities:\n\n', y_test_pred_prob[:3])
```

```
Train predictions for probabilities:

 [[2.64995300e-308 0.00000000e+000 1.27585873e-262 1.00000000e+000
  8.34182527e-240 8.75299845e-146 0.00000000e+000 1.08904980e-142
```

```
  6.30404646e-100 6.17490354e-213]
 [4.73210253e-280 1.71548669e-016 3.29550062e-240 1.94603482e-049
  2.82671254e-095 5.10285070e-018 6.06623725e-046 0.00000000e+000
  1.00000000e+000 1.31856143e-167]
 [0.00000000e+000 1.00000000e+000 3.71870946e-154 2.02512503e-184
  3.13217191e-124 0.00000000e+000 1.45531192e-308 6.60447224e-269
  7.21689339e-189 5.82585452e-159]]

Test predictions for probabilities:

 [[0.00000000e+000 0.00000000e+000 0.00000000e+000 3.24672851e-198
  7.56237323e-238 2.71834851e-292 0.00000000e+000 1.00000000e+000
  1.97727447e-208 4.99936734e-177]
 [3.61093354e-315 0.00000000e+000 4.44404745e-206 0.00000000e+000
  1.00000000e+000 1.58179754e-262 1.56999398e-227 4.80358703e-213
  2.56850684e-237 9.88053275e-098]
 [0.00000000e+000 1.00000000e+000 1.39571404e-126 6.76842535e-103
  5.71657752e-264 1.49898565e-109 2.68590752e-197 1.87687924e-299
  3.73110200e-088 1.01522551e-187]]
```

[26]:
```
y_train_pred = y_train_pred_prob.argmax(axis=1)  # Finding the predicted class␣
 ↪for train predictions
y_test_pred = y_test_pred_prob.argmax(axis=1)  # Finding the predicted class for␣
 ↪test predictions

# Presenting the predicted classes for the 1st 10 instances of train

print('Predicted class for train: ', y_train_pred[:10],'\n')

# Presenting the predicted classes for the 1st 10 instances of test

print('Predicted class for train: ', y_test_pred[:10])
```

```
Predicted class for train:  [3 8 1 8 9 1 4 7 3 3]

Predicted class for train:  [7 4 1 2 8 6 1 9 2 4]
```

[27]:
```
# Calculating the train and test accuracy scores of the model

from sklearn import metrics    # Importing 'metrics' from sklearn

train_score = metrics.accuracy_score(y_train, y_train_pred) # train accuracy

test_score = metrics.accuracy_score(y_test, y_test_pred)    # test accuracy

print('Train accuracy score of the model is ', round(train_score, 5),'\n')
print('Test accuracy score of the model is ', round(test_score, 5))
```

```
Train accuracy score of the model is  1.0

Test accuracy score of the model is  0.86867
```

At this point, you should be able to see signs of overfitting!

Question: Present two reasons as to why the above logistic regression classifier has a high level of variance.

To cure the overfitting, we can regularize the classifier. scikit-learn has an option to easily include a penalty term. Note the hyperparameter C is the *inverse of the regularization strength*. This means that a high/low value of C induces a weak/strong penalty term.

[28]:
```
# Instantiating logistic regression with 'L2' regularization

logreg_pen = LogisticRegression(penalty = 'l2', max_iter = 10000, C=10e-7) #␣
 ↪Including a strong L2 penalty

import timeit

start = timeit.default_timer()
logreg_pen.fit(X_train, y_train)  # Fitting the train data to 'logreg'
stop = timeit.default_timer()

print('Training time: %s sec' %(stop-start))
```

```
Training time: 4.856047946 sec
```

[29]:
```
# Finding the predicted class for the train and test subsets

y_train_pred = logreg_pen.predict_proba(X_train).argmax(axis=1)
y_test_pred = logreg_pen.predict_proba(X_test).argmax(axis=1)
```

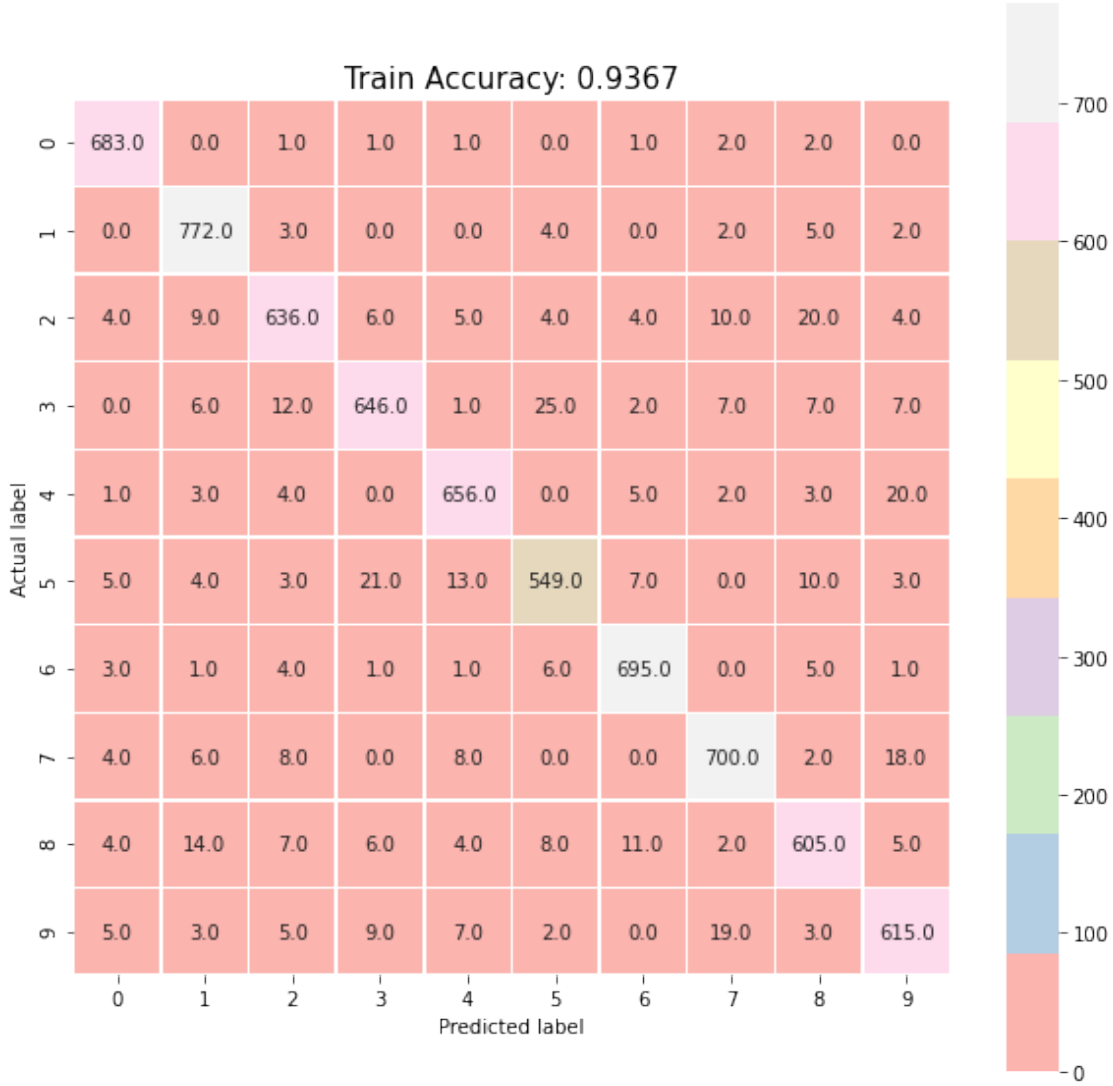
[30]:
```
train_score = metrics.accuracy_score(y_train, y_train_pred) # train accuracy

test_score = metrics.accuracy_score(y_test, y_test_pred)    # test accuracy

print('Train accuracy score of the model is ', round(train_score, 5),'\n')
print('Test accuracy score of the model is ', round(test_score, 5))
```

```
Train accuracy score of the model is  0.93671

Test accuracy score of the model is  0.91833
```

The accuracies now present a much better situation!

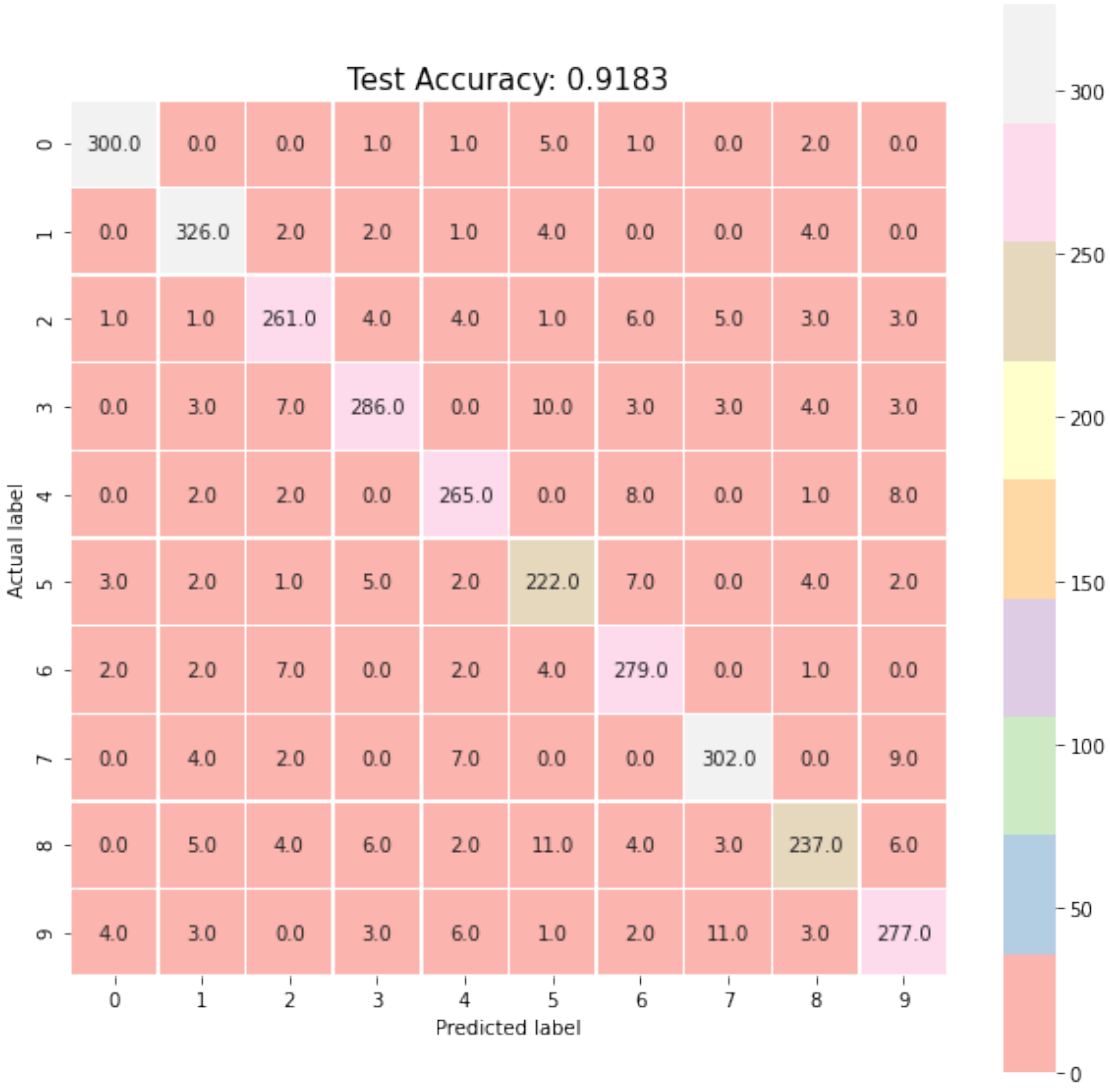
[31]:
```
# Computing the confusion matrix for the train subset

from sklearn.metrics import confusion_matrix, classification_report
```

```
train_c_matrix = confusion_matrix(y_train, y_train_pred)  # Compute the train␣
 ↪confusion matrix

plt.figure(figsize=(10,10))
sns.heatmap(train_c_matrix, annot=True, fmt=".1f", linewidths=.5, square = True,␣
 ↪cmap = 'Pastel1');
plt.ylabel('Actual label');
plt.xlabel('Predicted label');
all_sample_title = 'Train Accuracy: {0}'.format(round(train_score, 4))
plt.title(all_sample_title, size = 15);
plt.show()

# Print the train classification report

print('Classification Report for the Train Subset: \n\n',␣
 ↪classification_report(y_train, y_train_pred))
```



```
Classification Report for the Train Subset:

              precision    recall  f1-score   support

           0       0.96      0.99      0.98       691
           1       0.94      0.98      0.96       788
           2       0.93      0.91      0.92       702
           3       0.94      0.91      0.92       713
           4       0.94      0.95      0.94       694
           5       0.92      0.89      0.91       615
           6       0.96      0.97      0.96       717
           7       0.94      0.94      0.94       746
           8       0.91      0.91      0.91       666
           9       0.91      0.92      0.92       668

    accuracy                           0.94      7000
   macro avg       0.94      0.94      0.94      7000
weighted avg       0.94      0.94      0.94      7000
```

[32]:
```
# Computing the confusion matrix for the test subset

test_c_matrix = confusion_matrix(y_test, y_test_pred)  # Compute the test␣
 ↪confusion matrix

plt.figure(figsize=(10,10))
sns.heatmap(test_c_matrix, annot=True, fmt=".1f", linewidths=.5, square = True,␣
 ↪cmap = 'Pastel1');
plt.ylabel('Actual label');
plt.xlabel('Predicted label');
all_sample_title = 'Test Accuracy: {0}'.format(round(test_score, 4))
plt.title(all_sample_title, size = 15);
plt.show()

# Print the train classification report

print('Classification Report for the Test Subset: \n\n',␣
 ↪classification_report(y_test, y_test_pred))
```



Classification Report for the Test Subset:

	precision	recall	f1-score	support
0	0.97	0.97	0.97	310
1	0.94	0.96	0.95	339
2	0.91	0.90	0.91	289
3	0.93	0.90	0.91	319
4	0.91	0.93	0.92	286
5	0.86	0.90	0.88	248
6	0.90	0.94	0.92	297
7	0.93	0.93	0.93	324
8	0.92	0.85	0.88	278
9	0.90	0.89	0.90	310

```
    accuracy                           0.92      3000
   macro avg       0.92      0.92      0.92      3000
weighted avg       0.92      0.92      0.92      3000
```

Question: How do you evaluate the above multinomial classifier?

Question: The greatest off-diagonal element of the above confusion matrix is 11. Find the (off-diagonal) spots in the confusion matrix with value 11. Why do you think you observe the greatest off-diagonal entries at these spots?

[ ]: