CSDS-391 Intro to A.I., project 2

Nicolas Slavonia, njs140

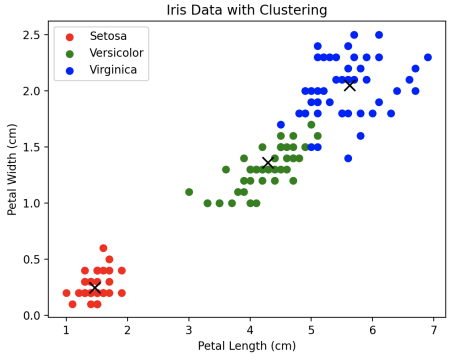
November 2022

Code Design

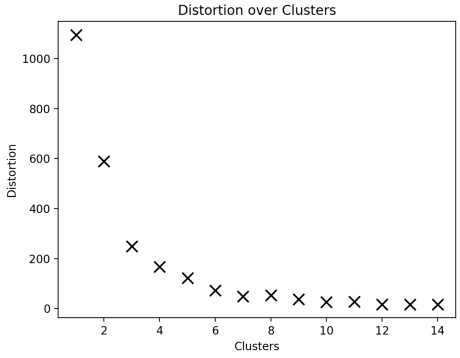
My code is broken up into 6 sections. I have my main python file which is called *P*2.*py*. This file has all the necessary commands to run every question. Each question are in their own file called *P*2*Q*1.*py*, *P*2*Q*2.*py*, *P*2*Q*3.*py*, *P*2*Q*4.*py*, *P*2*Q*5.*py* respectively.

Q1

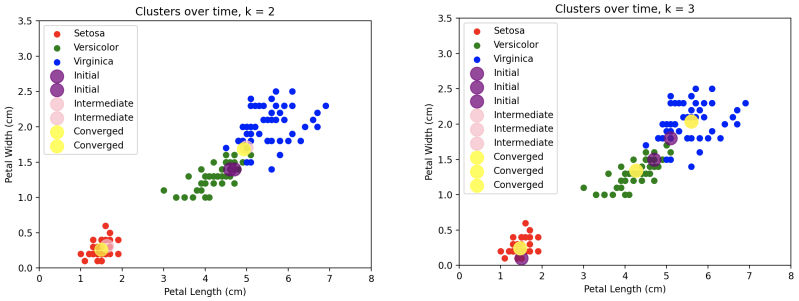
a) Equation is in the code, but I used two functions to calculate it. The first distances between the data points, the second updates the clusters.



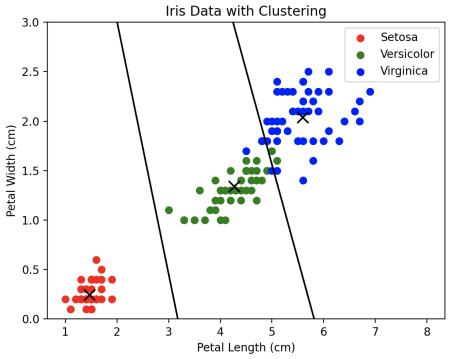
b) Plot that shows distortion (D) decreases as number of clusters increases.



c) Plot that shows the convergence of the clusters on the data (sorry for the weird legend).

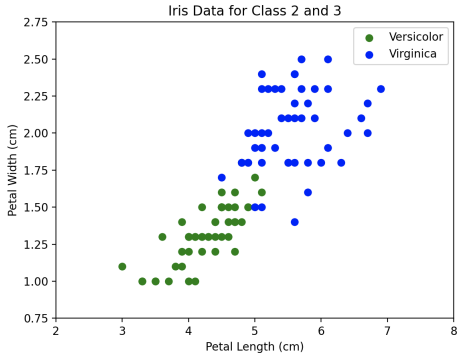


d) I found the lines by first finding the coordinates of the clusters, then using those coordinates, I plotted lines in between them with a slope perpendicular to their lines of intersection.



Q2

a) Part a is to make a plot of the second and third classes.



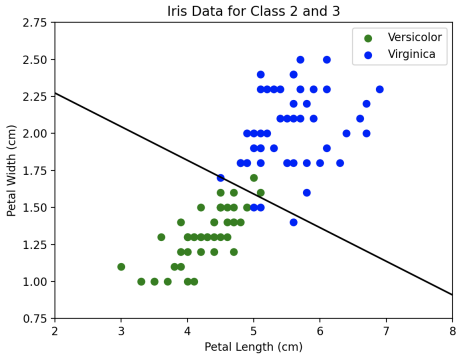
b) The sigmoid function is:

$$\frac{1}{1+e^{-wx+b}}$$

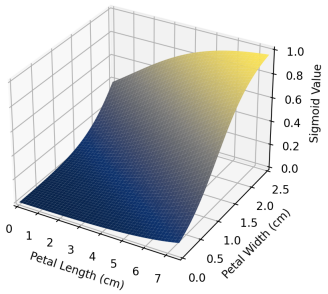
Where wx is the dot product between w and x. In code, this looks like:

```
1    def getSigmoid(w1, w2, bias, x1, x2):
2        return 1 / (1 + np.exp(-(w1*x1 + w2*x2) + bias))
```

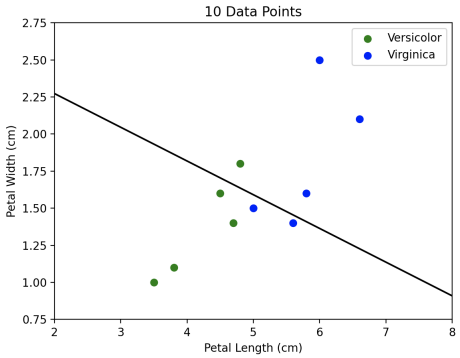
c) Using $w_1 = 0.5, w_2 = 2.2, b = 6$, the plot looks like:



d) The plots of the weights look like:



e) I picked 10 points that spread out the plot. I calculate the sigmoid value for each point, and determine if it is a versicolor (< 0.5) or virginica (> 0.5). The code also prints the actual values calculated for each point.



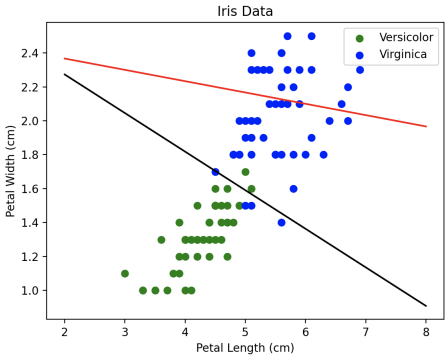
Q3

a) For every data point in the iris set, calculate

$$\frac{1}{n}\sum_{i=1}^{n}(class(x_i) - sigmoid(x_i))^2$$

```
1    sigmoid = 1 / (1 + np.exp(-(w1*x1 + w2*x2) + bias))
2    if species[i] = "versicolor"
3        real = 0
4    else
5        real = 1
6    sum = sum + (real - sigmoid)**2
7    return sum/#of data points
```

b) The good weights are $w_1 = 0.5, w_2 = 2.2, b = 6$ (error = 0.09001), the bad weights are $w_1 = 0.2, w_2 = 3, b = 7.5$ (error = 0.18133). I plotted both lines over the data like question 2.



c) Take the gradient of $(y(x) - \sigma(-w^T x + b))^2$ with respect to w.

$$\frac{\partial}{\partial w}(y(x)-\sigma(-w^T x+b)^2 = 2(y(x)-\sigma(-w^T x+b))\frac{\partial}{\partial w}(y(x)-\sigma(-w^T x+b))$$

$$2(y(x) - \sigma(-w^T x + b))(-\sigma'(-w^T x + b)\frac{\partial}{\partial w}(-w^T x + b))$$

$$2(y(x) - \sigma(-w^T x + b))(-\sigma(-w^T x + b))(1 - \sigma(-w^T x + b))(-x)$$

$$-2(\sigma(-w^T x + b) - y(x))\sigma(-w^T x + b)(1 - \sigma(-w^T x + b))x$$

d) The scalar form is derived above for each w:

$$\frac{\partial}{\partial \omega_0} = -2(\sigma(-w^T x + b) - y(x))\sigma(-w^T x + b)(1 - \sigma(-w^T x + b))$$

$$\frac{\partial}{\partial \omega_1} = -2(\sigma(-w^T x + b) - y(x))\sigma(-w^T x + b)(1 - \sigma(-w^T x + b))(-x_1)$$

$$2(\sigma(-w^T x + b) - y(x))\sigma(-w^T x + b)(1 - \sigma(-w^T x + b))x_1$$

$$\frac{\partial}{\partial \omega_2} = -2(\sigma(-w^T x + b) - y(x))\sigma(-w^T x + b)(1 - \sigma(-w^T x + b))(-x_2)$$

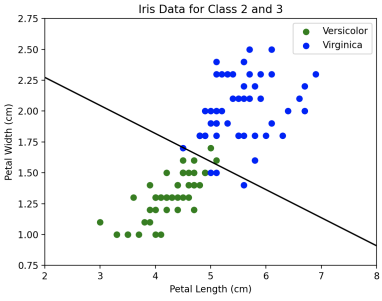
$$2(\sigma(-w^T x + b) - y(x))\sigma(-w^T x + b)(1 - \sigma(-w^T x + b))x_2$$

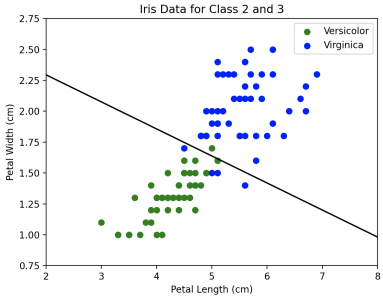
The vector form is the scalar forms put together:

$$\frac{\partial}{\partial \omega}(y(x) - \sigma(-w^T x + b))^2 = \begin{Bmatrix} \frac{\partial}{\partial \omega_0} \\ \frac{\partial}{\partial \omega_1} \\ \frac{\partial}{\partial \omega_2} \end{Bmatrix} = \begin{Bmatrix} -2(\sigma(-w^T x + b) - y(x))\sigma(-w^T x + b)(1 - \sigma(-w^T x + b)) \\ 2(\sigma(-w^T x + b) - y(x))\sigma(-w^T x + b)(1 - \sigma(-w^T x + b))x_1 \\ 2(\sigma(-w^T x + b) - y(x))\sigma(-w^T x + b)(1 - \sigma(-w^T x + b))x_2 \end{Bmatrix}$$

e) With a step size of 0.01, weights = weights - stepSize*gradient: (left photo is before the step). Initial weight = [0.5 2.2 6], new weight = [0.482 2.201 6.013].

```
1    for i = 50 to 150
2    if species[i] = "versicolor":
3        y = 0
4    else
5        y = 1
6    sigmoid = 1 / (1 + math.exp(-(w1*x1 + w2*x2) + b))
7    grad[0] = grad[0] + (sigmoid - y) * sigmoid * (1 - sigmoid) * x1
# for w1
8    grad[1] = grad[1] + (sigmoid - y) * sigmoid * (1 - sigmoid) * x2
# for w2
9    grad[2] = grad[2] - (sigmoid - y) * sigmoid * (1 - sigmoid)
# for bias
```

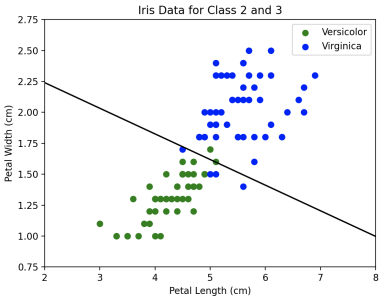


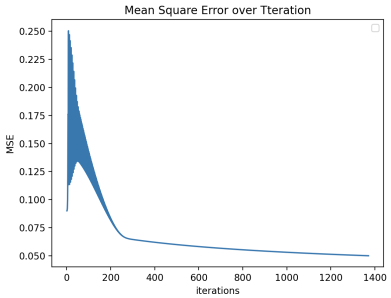


Q4

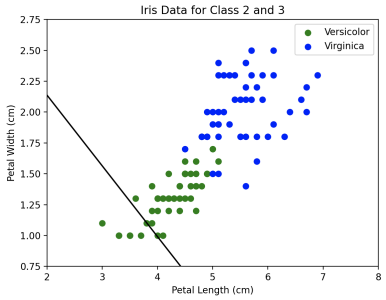
a)
```
converged = True
while converged:
    gradient = P2Q3.gradient(petalL, petalW, weights, species)
    weights = weights - stepSize*gradient
    if np.linalg.norm(gradient) < .2:
        converged = False
```

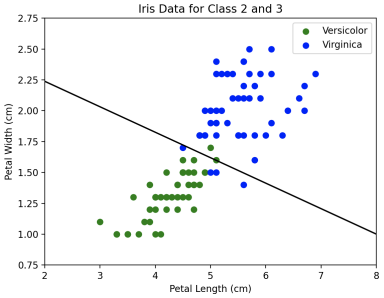
b) Plots for the final decision boundary and the mean square error:

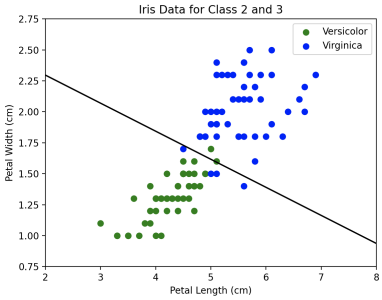


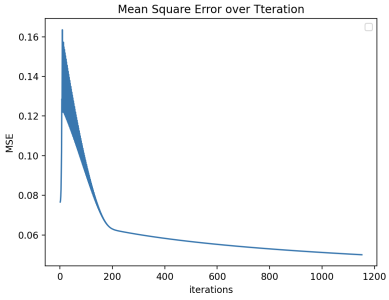


c) Plots for initial (top left), middle (top right) and final (bottom left) weights along with the mean square error plot.









d) I wanted a gradual conversion. If it was too big, it would jump all around and never converge. If it were too small, it would take forever too converge. But I went with a smaller step-size because the program is pretty fast.

$$step - size = 0.01$$

e) Looking at the MSE plot, once the MSE gets close to 0.05, it starts to plateau. The smaller the MSE the better, but this plateau shows that it will take a long time to be perfect. So I had it quit once the MSE was below 0.05. Ideally, stop when MSE = 0.

$$MSE < 0.05$$

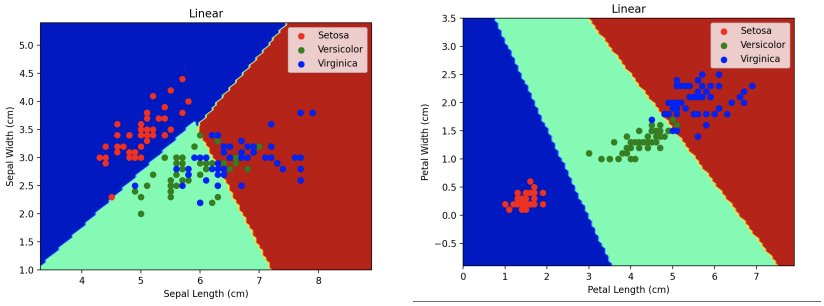
Q5

I do two things for extra credit. The first is I analyzed different types of decision boundaries. I compare linear, radial basis function (RBF) and polynomial boundaries. I do this using the *sklearn* toolbox. Each plot shows the data and the backgrounds are the decision boundaries. The second part I look at is testing different step sizes. I use the *keras* toolbox to run 250 epochs and monitor the accuracy while changing the step size.

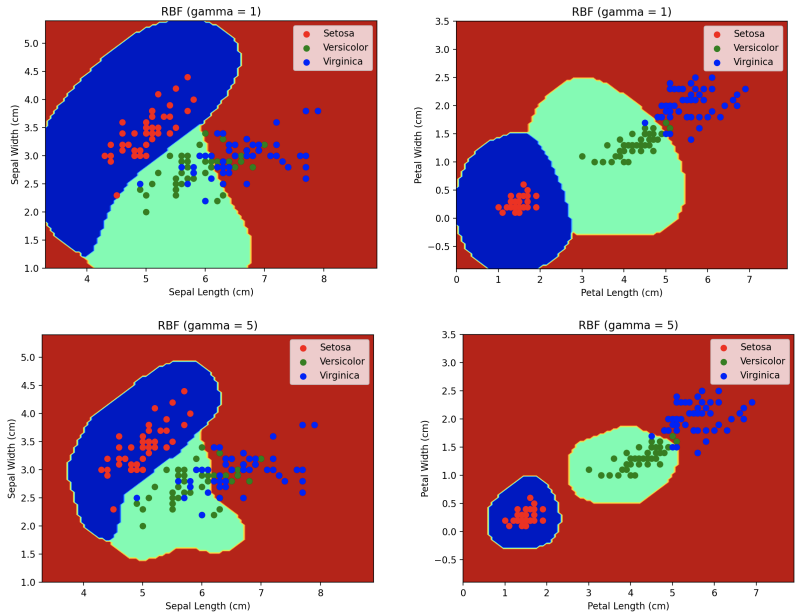
a) My code uses *sklearn* and $DecisionBoundaryDisplay.from_estimator()$ to calculate the decision boundaries. On line 1 through 6, I set up the model (which decision boundary to use) and the data. In the for loop, I run their method to calculate the decision boundary, and then plot it.

```
1    iris = datasets.load_iris()
2    data = np.concatenate(([petalL], [petalW])).T
3    if choice == True:
4        data = np.concatenate(([sepalL], [sepalW])).T
5    models = [svm.SVC(kernel="linear"), svm.SVC(kernel="rbf", gamma=1),
 svm.SVC(kernel="poly", degree=5, gamma=3)]
6    titles = ["Linear","RBF (gamma = 1)",
 "Polynomial (5th Degree, gamma = 3)"]
7    for i = 0 to 3
8    DecisionBoundaryDisplay.from_estimator(models[i].fit(data,
 iris.target),data,cmap=mpl.cm.jet,response_method="predict")
9    plot()
```

The first two plots are linear plots showing linear boundaries for all three classes over the sepal and pedal data sets.

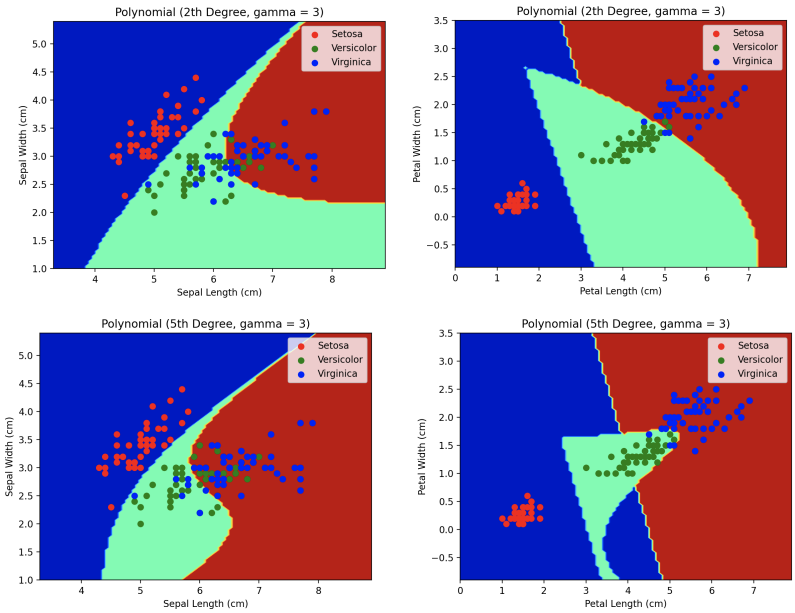


The second plot shows the decision boundary we found above in question 1 and 4 (red-green border, they are similar). The next plots show the RBF plots with gamma level 1 and 5. The gamma levels can represent the precision of the boundaries.



For low levels of gamma, the boundaries mimic the linear boundaries. In fact, for very low levels of gamma (0.1), the boundaries are nearly identical to linear boundaries. Otherwise, a base petal type is chosen and the other types are given their own area. In this case, Virginica type is the base.

The next plots show the polynomial decision boundaries for different levels of degree. These plots also take a value of gamma, but I set it constant to 3 so I could focus on the degree changes.



These plots also resemble the linear boundaries. I can conclude that the difference between the three methods mostly changes in locations that are far away from the clusters of data. Whereas the boundaries that are very close to the data are nearly the same. I personally think for this data, the RBF decision boundaries are the most accurate because they mold around the data. Although, RBF may not work well with outliers.

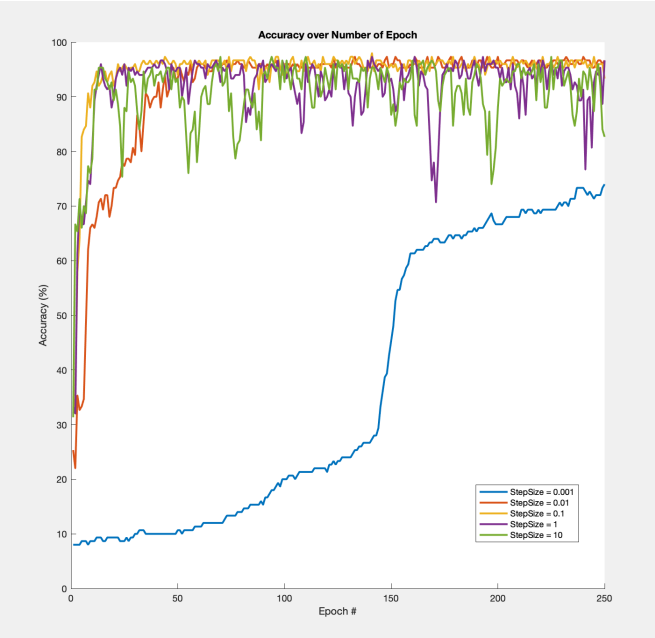
b) I use *keras* to calculate the accuracy of decision boundaries for 250 gradient modifications.

```
1    fullData = concat(sepalL, sepalW, petalL, petalW)
2    temp = np.array([1, 0, 0])
3    for i in range(150):
4        Y.append(temp)
5        if i == 50:
6            temp = np.array([0, 1, 0])
7        if i == 100:
8            temp = np.array([0, 0, 1])
9    Y = np.asarray(Y)
10
11   model = keras.models.Sequential()
12   model.add(keras.Input((fullData[0].size)))
13   model.add(keras.layers.Dense(3, activation='softmax'))
14
15   model.compile(optimizer=keras.optimizers.Adam(learning_rate = stepSize),
 loss=keras.losses.categorical_crossentropy, metrics=['accuracy'])
16   model.fit(fullData, Y, epochs = 500)
```

Line 1 through 9 set up the data. Y is used to identify the different classes. Lines 11 through 13 give *keras* the model I want to use and line 15 and 16 compile and print the outcome to the screen. Below is an example of the output when stepsize = 0.01

```
Epoch 1/250
5/5 [==============================] - 0s 2ms/step - loss: 2.5633 - accuracy: 0.6000
Epoch 2/250
5/5 [==============================] - 0s 1ms/step - loss: 2.1086 - accuracy: 0.6600
Epoch 3/250
5/5 [==============================] - 0s 1ms/step - loss: 1.7014 - accuracy: 0.6600
Epoch 4/250
5/5 [==============================] - 0s 1ms/step - loss: 1.2589 - accuracy: 0.6600
Epoch 5/250
5/5 [==============================] - 0s 1ms/step - loss: 0.8960 - accuracy: 0.6600
Epoch 6/250
5/5 [==============================] - 0s 1ms/step - loss: 0.7100 - accuracy: 0.7733
Epoch 7/250
5/5 [==============================] - 0s 1ms/step - loss: 0.7263 - accuracy: 0.7267
Epoch 8/250
5/5 [==============================] - 0s 1ms/step - loss: 0.7279 - accuracy: 0.6267
Epoch 9/250
5/5 [==============================] - 0s 1ms/step - loss: 0.6802 - accuracy: 0.8333
Epoch 10/250
5/5 [==============================] - 0s 1ms/step - loss: 0.6465 - accuracy: 0.7667
Epoch 11/250
5/5 [==============================] - 0s 1ms/step - loss: 0.6325 - accuracy: 0.7000
Epoch 12/250
5/5 [==============================] - 0s 1ms/step - loss: 0.6164 - accuracy: 0.7133
Epoch 13/250
```

I am focusing my attention on the accuracy given by each epoch. The plot below is a plot of 5 different step-sizes: 0.001, 0.01, 0.1, 1, 10.



This plot shows us how the smaller step-sizes take a very long time to converge. In fact, a step-size of 0.001 didn't fully converge over 250 iterations. While the larger step-sizes, 1 and 10, did converge but were very likely to overshoot which is what those big dips are. The sweet spot was a step-size of 0.01 and 0.1. These converged and stayed converged. (This plot was made in MatLab, I wasn't able to plot the data in Python. The data was acquired by un-commenting $print(history.history$ and copying the data to MatLab).

Citations for extra credit:
$https://scikit-learn.org/stable/getting\_started.html$
$https://keras.io/guides/sequential\_model/$