Comp3223 : Coursework

Ixil Duncan Miniussi - idm1u19

2. Linear Regression with non-linear functions

We use the given function to create the data-set we will use for the majority of this section. with n=20 we generate :

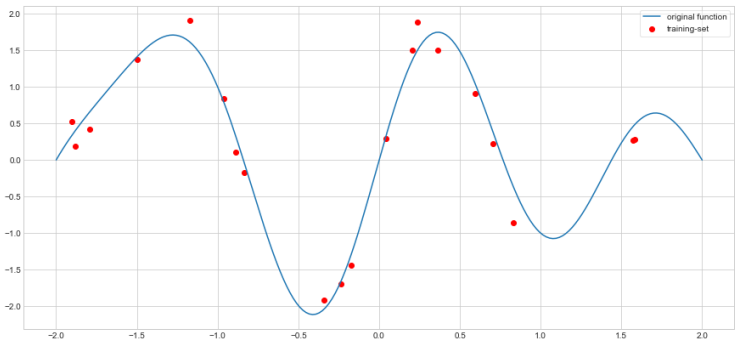

Figure 2.1.1 initial data-set generated from *generate_points(20, f, -2, 2)*

Although this report will not show every single line of code, 4 functions will be made use of heavily in this section.
For Polynomial curve fitting :

```
def polynomial_curve_fitting(p, l, X, Y) :

    j = np.arange(1, p+1, 1)
    I = np.eye(p+1)
    A = make_design(X, polynomial_basis_fn, j)
    AT = np.transpose(A)

    return np.dot(np.linalg.inv(AT@A + (l * I)), AT@Y)

def g(x, W):
    y = 0
    j = 0
    for w in W:
        y += w * (x ** j)
        j += 1
    return y
```

where the first function returns a vector of the correct weights $\vec{w}$ to minimize the loss function with a polynomial of degree P, using the analytical expression $\vec{w} = (A^T A + \lambda \mathbb{I}_{p+1})^{-1} A^T y$. And `g` creates said polynomial by following $y = w_0 1 + w_1 x + w_2 x^2 + \ldots + + w_p x^p$.

For Gaussian RBF fitting :

```
def gaussian_curve_fitting(p, l, X, Y) :

    if (p-1 <= 0) :
        j = [0]
    else :
        j = np.arange(-2, 2.001, 4/(p-1))

    A = make_design(X, gaussian_basis_fn, j)
    AT = np.transpose(A)

    f1 = np.linalg.inv(AT@A + l*np.identity((np.shape(A)[1])))
    return np.dot(np.linalg.inv(AT@A + l*np.identity((np.shape(A)[1]))), AT@Y)

def g_(x, W, sigma=0.2):
    y = 0
    p = len(W) - 1
    if (p-1 <= 0) :
        return np.zeros(len(x))
        j = []
    else :
        j = np.arange(-2, 2.001, 4/(p-1))

    for i in range(len(j)):
        y += W[i+1] * gaussian_basis_fn(x, j[i], sigma)
    return y
```

where similarly, the first function returns a vector of the correct weights, except it feeds the `make_design` function a list of 'centers' to use. These centers are evenly spread among our function, once we run `g_`, we find these same centers to create gaussians at each of these which we multiply by their respective weights and add up to create our fit. Visually speaking :

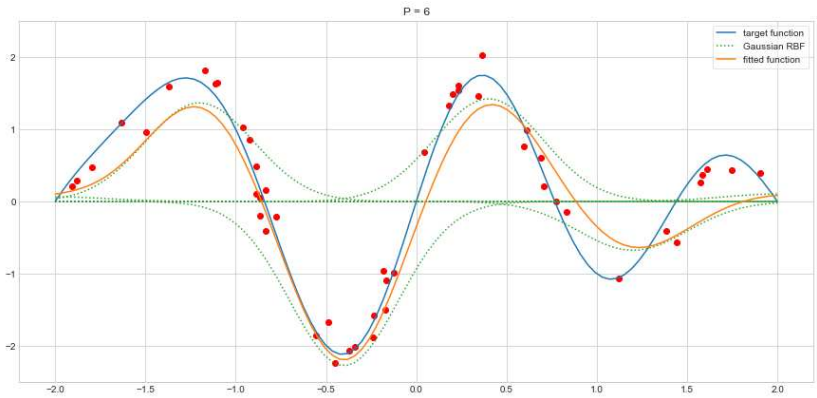

Figure 2.1.2 illustrative example of Gaussian RBF

RBF allows us for certain tuning that we need to be mindful of, the shape of our Gaussians depends on σ which we set to 0.3 by default. A large σ can be effective for lower values of P, as it allows for smoother transitions between the spaced out Gaussians, however you do need to be mindful that it isn't too high, or you will find yourself trying to use wide gaussians to fit a narrower function. Smaller σ can still work when given high P and with higher densities of points, as illustrated by the following examples:

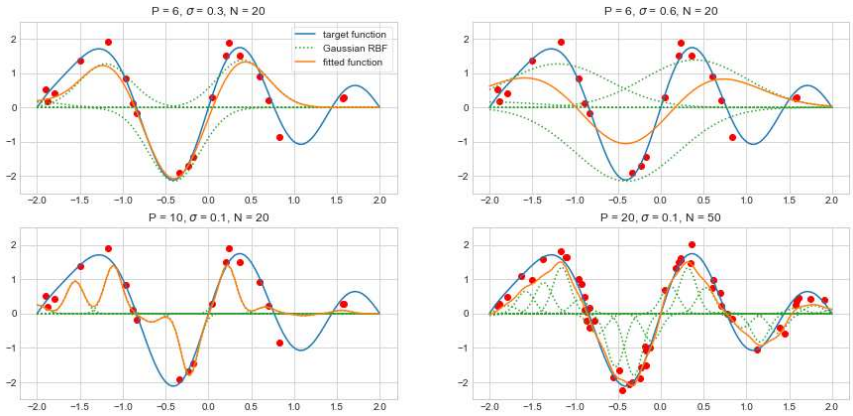

Figure 2.1.3 illustrative examples of how tuning Gaussian RBF curve fitting affects the end result

Depending on the function being fitted, this method can work great and be very efficient. The sweet-spot seems to be a σ that imitates the smoothness of the data being fitted, with centers facing local minimas and maximas.

Back to Polynomial fitting, we will keep our regularization parameter λ (`l`) to 0 and tune `p` to illustrate the effects of under and over-fitting

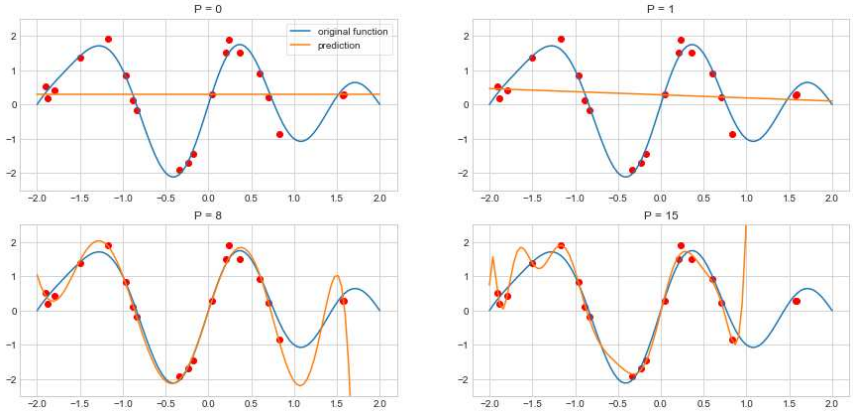

Figure 2.1.4 effects of varying P on polynomial curve fitting for the original Figure 2.1.1 data-set

This is relatively easy to understand. Remember that P corresponds to the degree of the polynomial we are creating, at P=0, we are essentially drawing a function $y = w_0$, and P=1 isn't doing much more with $y = w_1x + w_0$ (which on its own can be enough to find some direct correlations in certain data-sets). This time, our sweet-spot is P=8, we will justify it later but first lets take a final look at P=15 :

in this case, almost all the points are accounted for by the function, to near perfect accuracy. If we were to judge our prediction with this same set it would yield great results. However we come to a paradoxical issue of 'over-fitting', our function is trying too hard to attain these points and forgetting that the real target is the curve they were taken from. If we bring in more points, we will notice in fact that the function is performing worse than P=8 at fitting these new points. We can test this by having a training and testing set, we will train our algorithm against the training set, and then measure its performance against the testing set (which it was blind to until then), we will do the same with Gaussian RBF :

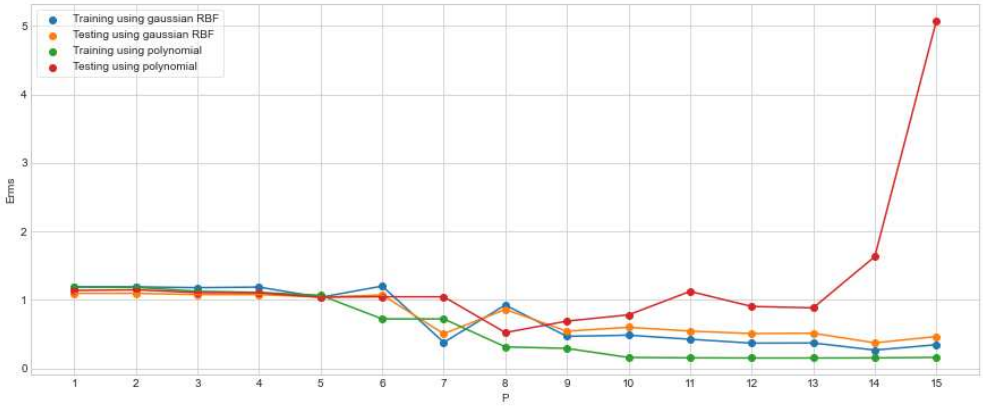

Figure 2.1.5 plotting performance of Gaussian RBF and polynomial curve fitting against varying P, using original data-set from Figure 2.1.1

This gives us a clearer picture of just how bad over-fitting can get with polynomial curve fitting, and also proves that P=8 is our best option since it yields the best results on our testing set. This same issue doesn't seem to arise with Gaussian RBF, as long as σ and λ are well chosen (so far we have been using $\ln \lambda = -1$ for Gaussian RBF curve fitting, and $\lambda = 0$ for polynomial curve fitting), a longer plot would show the same trend continuing.

To see why that is, we can simply look at the weights being generated at varying values of P.

using polynomial	P=0	P=1	P=6	P=15	using gaussian	P=1	P=2	P=9	P=15
w_0	0.12	0.12	-0.3	0.14	w_0	0.14	0.12	0.08	0.16
w_1		-0.07	2.48	8.23	w_1	-0.29	0.16	0.2	-0.02
w_2			0.88	-2.71	w_2		0.21	1.06	0.53
w_3			-3.02	-27.51	w_3			0.99	0.97
w_4			-0.07	4	w_4			-2.17	1.43
w_5			0.68	39.46	w_5			-0.2	-0
w_6			-0.03	0.7	w_6			1.58	-1.81
w_7				-36.22	w_7			-0.77	-2.18
w_8				-3.78	w_8			-0.08	0.08
w_9				21.12	w_9			0.24	1.58
w_{10}				2.37	w_{10}				0.63
w_{11}				-7.2	w_{11}				-0.24
w_{12}				-0.59	w_{12}				-0.81
w_{13}				1.29	w_{13}				-0.46
w_{14}				0.05	w_{14}				0.36
w_{15}				-0.09	w_{15}				0.13

Figure 2.1.6 table of weights used in Figure 2.1.5

We notice wild weights appearing with polynomial curve fitting, which we can confidently blame for the testing performance at P=15. $w_3 = -27.51, w_5 = 39.46, w_7 = 36.22$ and $w_9 = 21.12$ in particular.

However, as with most things in Machine Learning, a cure to over-fitting is sometimes simply 'more data to fit'

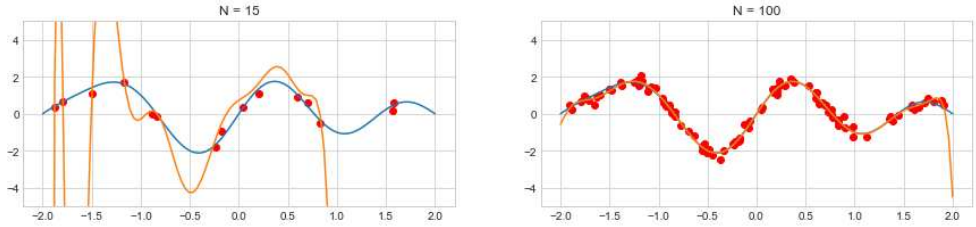

Figure 2.1.7 effects of using larger data-sets to combat over-fitting

Simply put, we are 'taming' our prediction. Since the issue we initially had is that the prediction curve favored reaching the training data at the expense of following the target curve. We simply gave it so many training points along said curve that reaching the first goal necessarily means reaching the other. This solution is obviously not *always* an option, but when it is, it works.

And finally, we will take look at the effects of the regularization parameter λ which we've so far ignored. So far, we've discussed 2 ways so far of avoiding over-fitting : lowering P, and finding more data. However there is a third method, which we have been using in regards to Gaussian RBF curve fitting for instance, and that is the regularization parameter. Effectively, λ seeks to add a penalty term to the error function in order to discourage the weights from reaching the wild values we saw in **Figure 2.1.6**, the higher the regularization term, the flatter the curve.

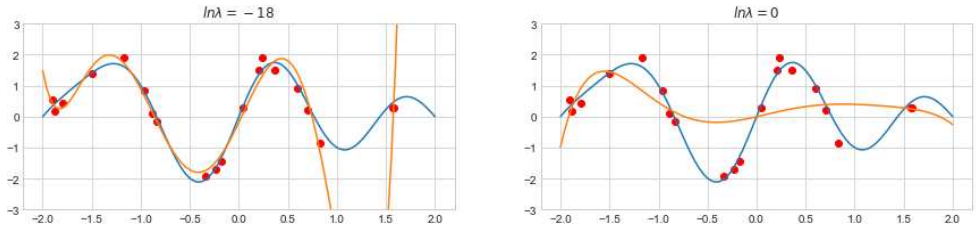

Figure 2.1.8 plots of P=8 fitted to our original data-set in Figure 2.1.1 with regularization parameter λ corresponding to lnλ = -18 and lnλ = 0

	lnλ=-inf	lnλ=-18	lnλ=0
w_0	0.02	0.02	0.31
w_1	7.21	7.21	1.35
w_2	1.79	1.79	-0.41
w_3	-18.63	-18.63	-1.01
w_4	-22.44	-22.4	-0.09
w_5	15.94	15.96	-1.02
w_6	62.31	62.21	0.06
w_7	-8.86	-8.89	-0.41
w_8	-73.74	-73.63	0.1
w_9	4.97	5.01	0.28
w_{10}	42.47	42.41	0.07
w_{11}	-2.31	-2.34	0.52
w_{12}	-11.58	-11.56	-0.05
w_{13}	0.71	0.72	-0.29
w_{14}	1.2	1.19	0.01
w_{15}	-0.1	-0.1	0.04

Figure 2.1.9 table of coefficients for P=15 with 3 different values for λ : -inf, -18 and 0

We can clearly see in this table for instance, the effects of lambda on the weights generated with higher values of P. We can also look at the effects of λ on over-fitting by plotting it against the root-mean-square Error (Erms) on our training and test sets using once again a high value for P (15). We now control our model complexity through higher and lower values for lambda. Although obviously, too much regularization eventually leads to a flat line, which we usually aren't looking for.

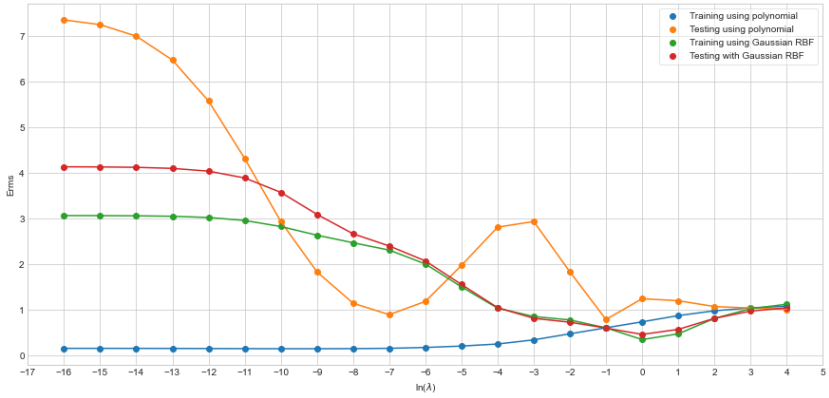

Figure 2.1.10 graph of the rms Error plotted against lnλ for P=15 polynomial and Gaussian RBF curve fitting

Lastly, we will take a quick and concise look at variance and bias. Imagine we're looking at the same prediction algorithm but with a variety of data-sets, all following our initial curve in **Figure 2.1.1**. These predictions will obviously vary from data-sets to data-set. We call the *squared bias* the extent to which the average prediction will vary from the target function, and we call *variance* the extent to which the individual predictions vary from their average. Therefor Variance is a good measure to how sensitive our curve-fitting function is to changes of data-sets. We plot 3 different examples of this 'thought experiment' with varying model complexity (which we now know we can control with λ)

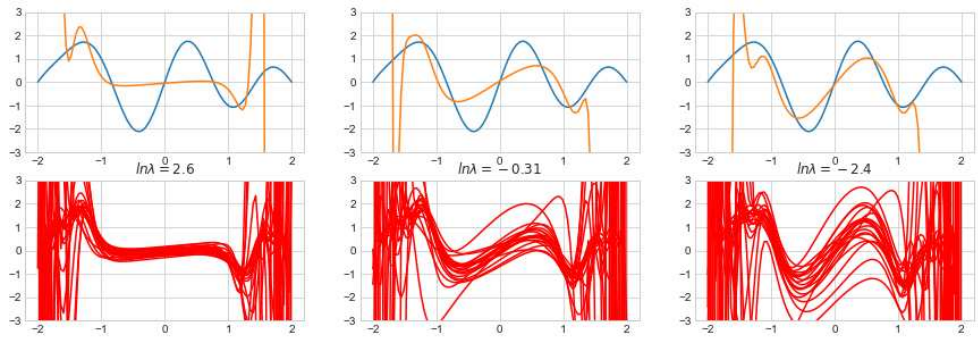

Figure 2.1.11 dependence of bias and variance on model complexity, the above row corresponds to the averages of the bottom row

We see here that lower model complexity leads to low variance but high bias. Whereas higher model complexity leads to high variance but low bias.

3. Classification

3.1 Data 1: separate 2 Gaussians

for our data sets, we generate a set of point from two 2-dimensional Gaussian distributions Σ_a ad Σ_b. numpy has a function `np.random.multivariate_normal()` that generates such data sets once given an average μ, a number of points n, and a covariance matrix Σ we generate ourselves through :

```
def S(p, s1, s2):
    return ((s1**2, p*s1*s2),(p*s1*s2, s2**2))
```

where s1 and s2 are the population variances (σ_1 and σ_2) and p is the population correlation. We generate 2 data sets of class a and b with $p = 0$, $n = 200$, $\sigma_1, \sigma_2 = 1$ and respectively $\mu_a = (-2, 0)^T$, $\mu_b = (2, 0)^T$.

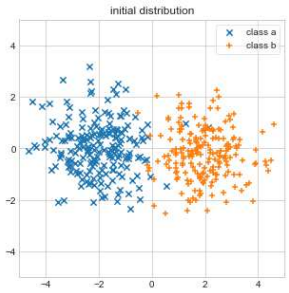

Figure 3.1.1 initial distribution generated from 2 2-dimensional gaussian distributions

To better understand the nature of class a and b, we decide to project them onto a lower dimension. In this case it isn't very useful because the 2d-dimension already makes it pretty clear which element belongs to which class, but this same concept becomes very practical once dealing with data-sets in higher dimensions. The goal is to simply remove a layer of complexity to our data-sets, while ideally preserving the distinctions between the two groups. We will arbitrarily pick two illustrative choices of w to project the data-sets onto: $\vec{w} = (1, 0)^T$ and $\vec{w} = (0, 1)^T$

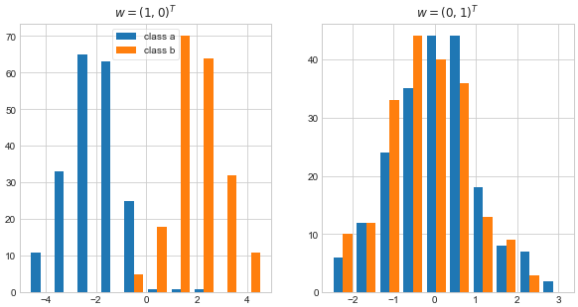

Figure 3.1.2 projected distribution of class a and b on 2 perpendicular vectors

In this example, we display the projections as histograms, a higher bar indicates a higher density of points. It becomes pretty clear which vector $\vec{w}$ makes for a clearer projection. In our first choice, what looks like 2 distinct Gaussian distributions reveal themselves, with means pretty much unmoved ($\mu_a . x = -2$, $\mu_b . x = 2$). Whereas in the second choice of $\vec{w}$, those same gaussians become confused, you can imagine that without the color coding, it would become pretty difficult to distinguish between data from one class to another.

Hopefully it's clear by now that $\vec{w}_1$ and $\vec{w}_2$ weren't chosen arbitrarily, the first one is a vector that goes through both μ_a and μ_b, while the second is perpendicular to that same line. This works great for two reasons :

1. We only have 2 classes, if there were 3, drawing a line between all means wouldn't be so easy (we will look into this in section 3.2).
2. Our distributions, although random, follow neat, perfectly symmetrical and spread out gaussians. You could imagine certain distributions where the separation might not be so clear.

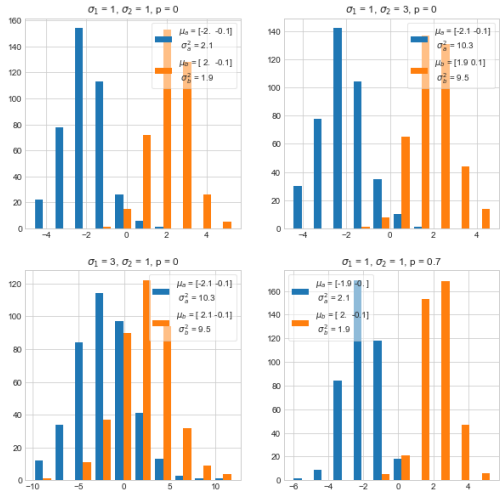

Figure 3.1.3 histogram of the effects of varying the covariance matrices a and b on the resulting projections

In the top left, we have our previous histogram as comparison.

The top right histogram is similar except we increased σ_2 by a factor of 3. Yet the projected distribution remains the same. Simply put σ_2 affects the spread of our distribution along the y axis (with $p = 0$) which we essentially get rid of once we project it along $\vec{w} = (1, 0)^T$. The bell curve is therefor unaffected in any meaningful way.

The bottom left histogram is the consequence of increasing σ_1, this σ corresponds to the spread of our distribution along the x axis. A higher spread might not affect the location of our class means, but it makes classifying the data more difficult, we find a lot more overlap between the 2 sets and the peaks are lower. Simply drawing a line inbetween the 2 means will yield only above average accuracy, and a statistical approach to classification will probably be more accurate.

Finally, the bottom right histogram demonstrates what happens when you tune the p parameter. The resulting histogram is relatively unchanged, but $\vec{w} = (1, 0)^T$ is by no means are best choice.

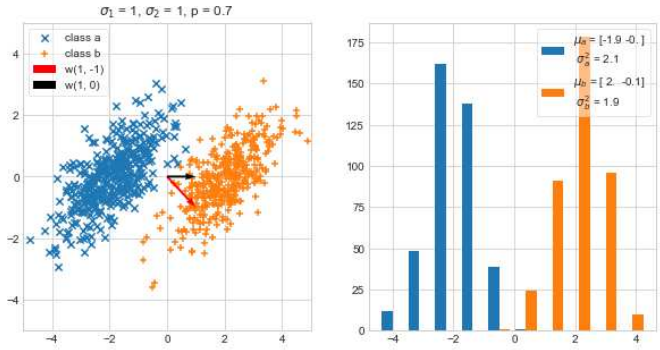

Figure 3.1.4 proposing an alternative vector w to to improve the projection of the bottom left histogram in Figure 3.1.3

Notice the higher peaks and greater separation in the histogram with a new choice of $\vec{w} = (1, -1)^T$. This choice becomes obvious when looking at the shape of our new data-sets. We therefor understand that simply looking at the means might not always be the best starting point for projecting data-sets onto lower dimensions.

We've looked at different distribution and gave guesses as to how easy they would be to work with. But how do we *know* a good projection?

Fisher's ratio $f(\vec{w})$ answers that question:

$F(w) \triangleq \frac{(\mu_a - \mu_b)^2}{\frac{n_a}{n_a+n_b}\sigma_a^2 + \frac{n_b}{n_a+n_b}\sigma_b^2}$

The goal is to maximize said ratio, we notice that there are 2 ways to do so:

1. maximize the separation between the projected means
2. minimize the projected variance

We calculate Fisher's ratio in python using J() :

```
def J(w, Xa, Xb):
    na = len(Xa); nb = len(Xb)
    return (((np.linalg.norm(mu(Xa) - mu(Xb))) ** 2)/(((na/(na + nb)) * sigma(Xa)) + ((nb/(na + nb)) *
sigma(Xb))))

def mu(X):
    return np.mean(X, axis=0)

def sigma(X):
    r = 0; n = len(X); c = mu(X)
    for i in X:
        r += np.linalg.norm(i - c) ** 2
    return (r/n)
```

To visualize Fisher's ratio in action, we plot it's dependency on the direction of $\vec{w}$ by rotating a starting weight vector $\vec{w} = (1, 0)$ by angles of θ. We do so using functions `def R(theta): return ((np.cos(theta), -np.sin(theta)),((np.sin(theta), np.cos(theta))))` and `def W(theta): return (np.dot(R(theta), w0))`, where `w0` is our starting vector, and `theta` the rate at which we rotate it. When applied to our initial data-set [**Figure 3.1.1**] we find:

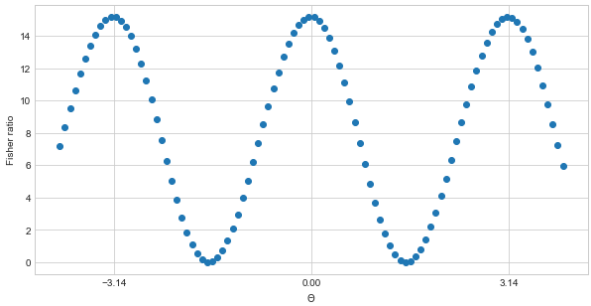

Figure 3.1.5 plot of Fisher's ratio against rotating vector w on original data-set from Figure 3.1.1

I.e. our initial assessment was correct, and did yield the best projection with $F(w_0) = 15.2$. We notice predictably that the Fisher's ratio repeats itself with every half rotation of our vector, since $\vec{w} = (1, 0)$ and $\vec{w} = (-1, 0)$ are on the same line. We also notice that our ratio is at it's worse once we rotate the correct vector by 90°, It might be tempting to generalize this property to all data-sets but we can't, a quick look at this same dependency when applied to our last data-set [**Figure 3.14**] will show a different story:

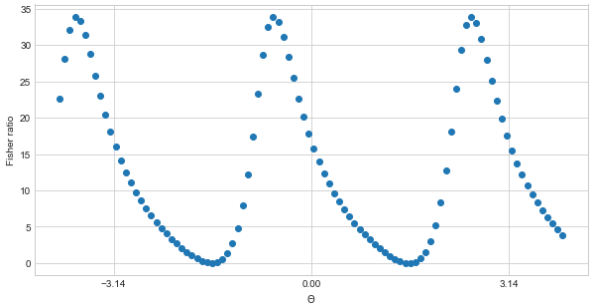

Figure 3.1.6 Dependence of Fisher's ratio on w with alternative data-set [Figure 3.1.4]

In this case, the highest point corresponds to $F(\vec{w}(2.55)) = F((0.78, -0.63)^T)$

3.2 Data 2: Iris data

Looking at this new data-set, we understand the need to perform LDA, since the 4 dimensions the data-set comes in are hard for us to visualize and conceive of.

Our goal is to find 2 weights to project the data onto while, again, conserving as much of the distinctions between classes as possible. Instead of finding these weights by direct search however, we use Fisher's generalized eigenvalue condition for optimal weights $\Sigma_B \vec{w} = \lambda \Sigma_W \vec{w}$, with Σ_B the between-class covariance and Σ_W the within-class covariance, which we find using:

```
def between_class_cov(X0, X1, X2) :
    mu_g = mu([mu(X0), mu(X1), mu(X2)])
    N = len(X)
    cov0 = (mu(X0) - mu_g) * np.reshape(mu(X0) - mu_g, (4,1)) * (len(X0)/N)
    cov1 = (mu(X1) - mu_g) * np.reshape(mu(X1) - mu_g, (4,1)) * (len(X1)/N)
    cov2 = (mu(X2) - mu_g) * np.reshape(mu(X2) - mu_g, (4,1)) * (len(X2)/N)
    return cov0 + cov1 + cov2

def within_class_cov(X0, X1, X2) :
    return np.cov(np.transpose(X0)) + np.cov(np.transpose(X1)) + np.cov(np.transpose(X2))
```

where `X0`, `X1`, and `X2` inputs correspond to our 3 sets for the classes of setosa, versicolor, and virginica.
We solve the generalized eigenvalue problem using `eigh` from `scipy.linalg` in `eigvals, eigvecs = eigh(Sb, Sw); eigvecs = np.transpose(eigvecs)`. By choosing `w = eigvecs[3]` for instance we can test that we have a correct solution by printing `(Sb @ w - (eigval * Sw @ w))` which does return an empty vector except for some small imprecision we can blame on numpy.

Having our 4 eigenvectors, we can now print the projection of our data-sets onto them, and look at the respective distributions given.

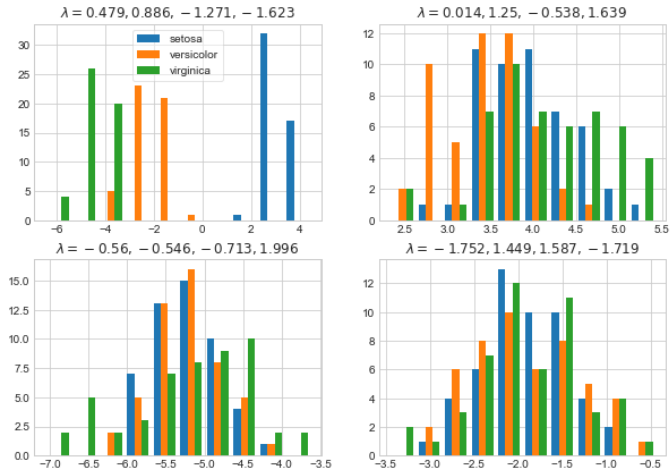

Figure 3.2.1 histograms of the projections along each of the eigenvectors of the Iris data, separated by class

We can notice a projection better than others in the top left corner, the class overlap is minimal, although not null (in particular between versicolor and virginica). The last bottom 2 distributions don't seem too useful, with major overlap between all classes. Same could be said with the second distribution, in the top right corner with the exception of a promising spike in versicolor, separate from the others, which while lacking on its own, could help making distinctions between versicolor and virginica, distinctions that our champion distribution doesn't fully address.

We therefore use the top 2 eigenvectors to project our data onto, we will use the results from our first one as our X axis, and the results from the second as our Y axis. This gives us the following 2d distribution :

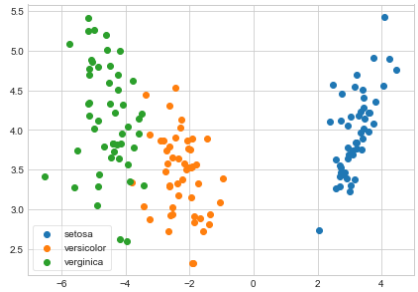

Figure 3.2.2 resulting projection of the Iris data-set onto the highest eigenvectors from Figure 3.2.1

We use softmax regression to classify said points, using `scikit-learn`'s train_test_split function on our dataset, with `test_size=0.33`, starting with a random weight and bias, we apply the following algorithm onto our training set:

```
[...]
for e in range(epochs):
        z = X@w + b
        y_ = softmax(z)

        y_clear = np.zeros((len(y), c))
        y_clear[np.arange(len(y)), y] = 1

        w_grad = (1/m) * np.dot(X.T, (y_ - y_clear))
        b_grad = (1/m) * np.sum(y_ - y_clear)

        w = w - lr*w_grad
        b = b - lr*b_grad

        loss = -np.mean(np.log(y_[np.arange(len(y)), y]))
        losses.append(loss)

    return w, b, losses

def softmax(z):
    exp = np.exp(z - np.max(z))

    for i in range(len(z)):
        exp[i] /= np.sum(exp[i])

    return exp
```

once ran with 4000 epochs and a learning rate `lr` of 0.9. Measuring the accuracy from the weights onto our testing set returns the following classification, with an accuracy of 0.96 on the testing set.

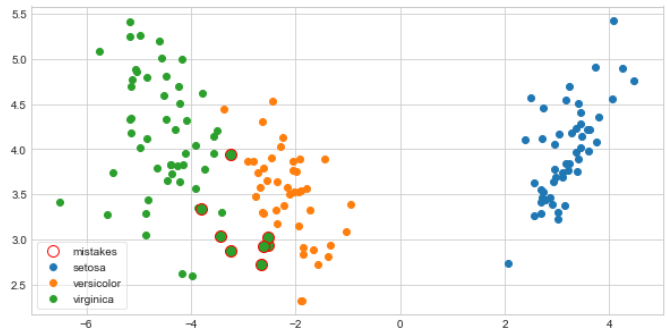

Figure 3.2.3 classified Iris data-set according to our softmax regression algorithm