

EECE 5644 Assignment 2

Eric Grimaldi

2020 November 5

Foreward on Code Libraries

For reference:

The code for this assignment was completed in *Python*. The *NumPy* library is used extensively for linear algebra and random variables. The *SciPy* library is used for a few linear algebra functions with improved functionality. The *Matplotlib* is used for data visualization. *TensorFlow2* provides the *tensorflow.keras* package which was used to quickly and efficiently implement the simple MLP required in Question 1.

Question 1

As per the assignment:

X is a 3-dimensional random vector drawn from the PDF below:

$$p(x) = \sum_{l=1}^C p_l(x|L=l)P(L=l),$$
$$P(L=l) = \frac{1}{C} \quad \forall l$$

There are $C = 4$ classes, $L = l$ signifies the true class label. The class prior are uniformly weighted. The class-conditional PDFs are Gaussians. As per the problem we select our own means and covariances, specified below:

$$\mu_1 = \begin{bmatrix} 1 \\ 1 \\ 0 \end{bmatrix} \quad \mu_2 = \begin{bmatrix} 1 \\ -1 \\ 0 \end{bmatrix} \quad \mu_3 = \begin{bmatrix} -1 \\ -1 \\ 0 \end{bmatrix} \quad \mu_4 = \begin{bmatrix} -1 \\ 1 \\ 0 \end{bmatrix}$$
$$\Sigma_l = \begin{bmatrix} 0.5 & 0 & 0 \\ 0 & 0.5 & 0 \\ 0 & 0 & 0.5 \end{bmatrix} \quad \forall l$$

For numerical results below, we generate 7 sets of data according to this distribution:

- D_{train}^{100} consisting of 100 samples and their labels, used for training
- D_{train}^{200} consisting of 200 samples and their labels, used for training

- D_{train}^{500} consisting of 500 samples and their labels, used for training
- D_{train}^{1000} consisting of 1000 samples and their labels, used for training
- D_{train}^{2000} consisting of 2000 samples and their labels, used for training
- D_{train}^{5000} consisting of 5000 samples and their labels, used for training
- D_{test}^{100000} consisting of 20000 samples and their labels, used for evaluation

To draw these samples, we first randomly choose a true label according to the class priors. Then we randomly draw a sample data point from the appropriate class-conditional PDF by using `numpy.random.multivariate_normal()`. After generating the data sets, they are saved for repeated future use.

First we construct the theoretically optimal classifier using the minimum-probability-of-error classification rule. As provided in *L02a...ClassifierDesign.pdf*, and explored in previous assignments, the classification rule is thus:

$$l = \underset{i}{\operatorname{argmin}} \Lambda P$$

$$\text{where } \Lambda = \begin{bmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{bmatrix} \quad \text{and } P = \begin{bmatrix} P(L = 1|x) \\ P(L = 2|x) \\ P(L = 3|x) \\ P(L = 4|x) \end{bmatrix}$$

After Bayes Rule, P becomes:

$$P = \frac{1}{p(x)} \begin{bmatrix} P(x|L = 1)P(L = 1) \\ P(x|L = 2)P(L = 2) \\ P(x|L = 3)P(L = 3) \\ P(x|L = 4)P(L = 4) \end{bmatrix}$$

After substitution and simplification, the decision rule becomes:

$$l = \underset{i}{\operatorname{argmin}} \begin{bmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{bmatrix} \begin{bmatrix} |\Sigma_1|^{-\frac{1}{2}} e^{-\frac{1}{2}(x-\mu_1)^T \Sigma_1^{-1} (x-\mu_1)} \\ |\Sigma_2|^{-\frac{1}{2}} e^{-\frac{1}{2}(x-\mu_2)^T \Sigma_2^{-1} (x-\mu_2)} \\ |\Sigma_3|^{-\frac{1}{2}} e^{-\frac{1}{2}(x-\mu_3)^T \Sigma_3^{-1} (x-\mu_3)} \\ |\Sigma_4|^{-\frac{1}{2}} e^{-\frac{1}{2}(x-\mu_4)^T \Sigma_4^{-1} (x-\mu_4)} \end{bmatrix}$$

With this classifier we evaluate D_{test}^{100000} in order to empirically estimate the minimum probability of error \hat{P}_e . As shown in *Figure1*, we find $\hat{P}_e = 15.12\%$.

For the remainder of Question 1, we construct a classifier using a Multi-Layer Perceptron (MLP) neural network in order to classify our data without knowledge of the true distribution.

As per the assignment, we construct a 2-layer MLP. *tensorflow.keras.Sequential* provides us with a sequential MLP from an input list of layers. *tensorflow.keras.layers.Dense()* provides us with fully connected dense layers of a input number of perceptrons which use an input activation function. As suggesting in class, biases for both layers are initialized as zeros (the default setting). Weights are randomly initialized using the Glorot Uniform distribution (also the default setting), which is effectively a uniform distribution where the range of possible values is determined by the number of inputs and outputs of the layer. We choose the first layer to have an arbitrary number of perceptrons p , and an Exponential Linear Unit (ELU) activation function. The number p will be selected later with K-Fold Cross Validation (KFCV). We choose the second layer to have $C = 4$ perceptrons, in order to produce a single output for each class conditional, and a Softmax activation function so that the outputs are properly constrained as class conditionals.

With the built in *Sequential.compile()* method, we select an optimization algorithm to perform gradient descent on our parameters, a loss function to use as a minimization objective, and some additional metrics. We select the Adam optimizer which was discussed in class. As per the assignment, we select Categorical Cross Entropy as our loss function (categorical in that it accepts arbitrarily many possible classes, as opposed to Binary Cross Entropy). In addition we also select the built in accuracy measurement as an additional metric, for KFCV, troubleshooting, and results reporting.

Thanks to *tensorflow.keras* The entire model is contained in the 6 line *get_model()* function, as shown in Figure 2. Note also that the *Sequential.fit()* method is designed to take an arbitrarily sized input at run time. The *Sequential.evaluate()* method expects the input to match the size of the input previously used to train the model.

With our model structure defined, we move on to hyper parameter selection with KFCV, and exploration of the effect of sample size on hyper parameter selection and model performance.

For each of the 6 training sets (generally: D_{train}) we use the following procedure:

For many possible choices of p , beginning with $p = 1$:

We split the training sample-label pairs into $K=10$ parts. Then for each k :

We get a fresh initialization of the model with p perceptrons in the first layer. We train the model using all training sample-label pairs except those in the k th part using the *Sequential.fit()* built in method. In order to ensure our model sees enough iterations to train fully, especially for the small datasets, we run 10 to 20 epochs. We also conduct several random initializations and choose the model with the best performance, in an effort to avoid local minimizers. Once training is complete, we evaluate the performance of the model on the k th part (which was left out of the fitting process) using the *Sequential.evaluate()* built in method. The evaluation classifies each sample and computes an empirical classification accuracy measurement, from which we calculate the empirical probability of classification error.

We take an average of empirical probability of classification error across all k parts for the given choice of p .

When the change in average error from increasing choice of p becomes negligibly small for several consecutive p , we stop trying additional p , and choose p^* to be the earliest p which achieved this relatively unchanging average error. Due to time constraints, we also set an upper limit on p at $p = 50$. This should be reasonable given the simplicity of our training

data, though it is certainly inelegant.

With our choice of p^* , we get a fresh model, train that model on the entire D_{train} , and then evaluate the model on the entire D_{test}^{100000} in order to compute our final \hat{P}_e for the model produce with that choice of D_{train} .

Finally we have 6 \hat{P}_e values each corresponding to a respective D_{train} . From our experiments we find \hat{P}_e decreases very quickly as the number of samples rises. As you can see in Figure 3, with only 500 samples \hat{P}_e is already approaching the performance of the theoretically optimal classifier. The number of perceptrons used to achieve \hat{P}_e also drastically decreases as the number of samples rises. For 5000 samples we see as few as 3 perceptrons! Meanwhile, even though 500 samples produces a good \hat{P}_e , we see that it required 46 perceptrons to do so. As is typical, we find that you may never have "too much" data, even if your results are beginning to look good.

We should take a moment before moving on in order to discuss the high number of perceptrons chosen in cases with very small training datasets. As you can see in Figure 4, in our experiments error decreases relatively slowly with increasing number of perceptrons for smaller datasets. One would expect that KFCV would disincentivize high numbers of parameters for small numbers of samples - however this was not the case. This could be due to several factors. The inherent noisiness random initialization in combination with a stochastic gradient descent optimize could be obscuring the parameter penalty inherent to KFCV. Also, due to the incredibly small number of parameters, it could be that even more epochs were needed to fully train the model, or perhaps we erroneously overtrained the model with too many epochs in an effort to overcome to inherent error of models trained with so few samples. If more time were available to us, these avenues would have been explored.

Question 2

As per the assignment:

X is a 7-dimensional random vector distributed according to a Gaussian with nonzero mean μ and nondiagonal covariance Σ . μ is arbitrarily chosen as $\mu = [1 \ 2 \ 3 \ 4 \ 3 \ 2 \ 1]^T$. The eigen values of Σ are arbitrarily chosen as $\{0.3, 0.5, 0.9, 1.1, 1.3, 1.5, 1.4\}$. Σ was randomly initialized from those eigen values using `scipy.stats.random_correlation.rvs()`, and then saved for repeated future use.

Z is a 7-dimensional random vector distributed according to a Gaussian with zero mean and covariance αI . Later in Question 2, we will explore a range of α .

V is a scalar random variable distributed according to a Gaussian with zero mean and unit variance.

For each of these random variables, we draw and save 100 samples for training (D_{train}) and 10000 samples for testing (D_{test}).

Y is the following function of X , which has Additive White Gaussian Noise (AWGN) corrupting both the input and output:

$$y = a^T(x + z) + v$$

We choose a arbitrarily to be $a = [4 \ 3 \ 2 \ 1 \ 2 \ 3 \ 41]^T$. From the previously drawn training and testing samples, we calculate corresponding Y .

In this problem we explore the impact of a model mismatch on the results of regression.

As per the assignment, we inappropriately decide the relationship between X and Y is:

$$y = w^T x + w_0 + v$$

We also believe that this model has parameters with a prior of a Gaussian with zero mean and covariance βI . Later in Question 2, we will choose β via KFCV. With MAP we are able to analytically determine w and w_0 from our model assumptions and a given set of training x - y pairs.

For convenience, we rewrite the model as:

$$y = \theta^T \phi + v$$

$$\text{where } \theta = \begin{bmatrix} w_0 \\ w \end{bmatrix} \text{ and } \phi = \begin{bmatrix} 1 \\ x \end{bmatrix}.$$

From the definition of MAP parameter estimation, with i.i.d. samples, and an assumption of independence between x_i and $\{w, w_0\}$ we know:

$$\hat{\theta} = \underset{\theta}{\operatorname{argmax}} \sum_{i=1}^N \ln p(y_i | \phi_i \theta) + \ln p(\theta)$$

In order to analytically calculate $\hat{\theta}$, we take the derivative of the objective, set it equal to zero, and solve for θ :

$$\left. \frac{d}{d\theta} [\ln p(\theta) + \sum_{i=1}^N \ln p(y_i | \phi_i \theta)] \right|_{\theta=\hat{\theta}} = 0$$

The two smaller expressions differentiate as follows:

$$\begin{aligned}\frac{d}{d\theta} \ln p(\theta) &= \frac{d}{d\theta} [\ln((2\pi)^{-\frac{d+1}{2}} |\beta I|^{-\frac{1}{2}} - \frac{1}{2\beta} \theta^T \theta)] = \frac{1}{\beta} \theta \\ \frac{d}{d\theta} \sum_{i=1}^N \ln p(y_i | \phi_i \theta) &= \frac{d}{d\theta} [\ln((2\pi)^{-\frac{1}{2}} - \frac{1}{2} (y_i - \theta^T \phi_i)^2)] = y_i \phi_i - (\phi_i \phi_i^T) \theta\end{aligned}$$

Which substitute into the original expression to give us:

$$-\frac{1}{\beta} \theta + \sum_{i=1}^N [y_i \phi_i - (\phi_i^T \phi_i) \theta] \Big|_{\theta=\hat{\theta}} = 0$$

This finally rearranges into:

$$\hat{\theta} = (\sum_{i=1}^N y_i \phi_i) (\frac{1}{\beta} I + \sum_{i=1}^N \phi_i^T \phi_i)^{-1}$$

Our entire training procedure for a given β and dataset is to solve this equation for the parameters of our model.

In order to select the hyperparameter β , we perform KFCV with $K = 10$ using D_{train} on a range of possible β from 10^{-3} to 10^3 (spaced logarithmically). The procedure is practically identical to the procedure used in Question 1, though we calculate $-2\log$ likelihood to use as our loss minimization objective.

Once the best β is selected, we calculate $\hat{\theta}$ one final time, and then evaluate the performance of our final model by calculating the $-2\log$ likelihood of D_{test} .

In order to test the impact of α on our selection of β as well as model performance, we perform this experiment 100 times for different α values from $10^{-3} \text{trace}(\Sigma)/7$ to $10^3 \text{trace}(\Sigma)/7$, spaced logarithmically. For each α , we draw a new D_{train} and D_{test} , compute a new β , and evaluate the model performance.

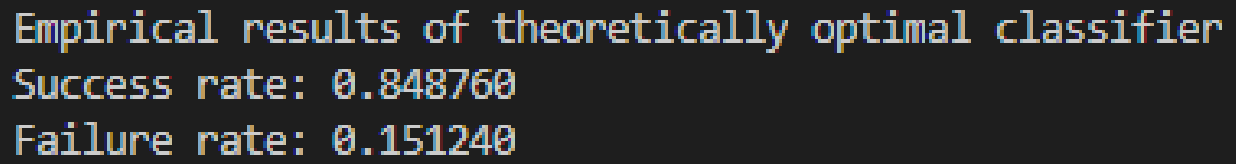
As shown in Figure 5, we find that as $\alpha \rightarrow \infty; \beta \rightarrow 0$.

As shown in Figure 6, we find that as $\alpha \rightarrow \infty; -2\ln p(D|\theta) \rightarrow \infty$. As input noise increases, the model mismatch becomes overwhelming. When input noise is somewhat low, such as in Figure 8, you can see the estimate of y "kind of" tracking along the true value of y . As the input noise factor trends toward zero, the tracking becomes actually rather accurate. Ultimately, the inappropriateness of the model mismatch is more or less proportional to the magnitude of noise ignored by the mismatch.

Appendix A: Code

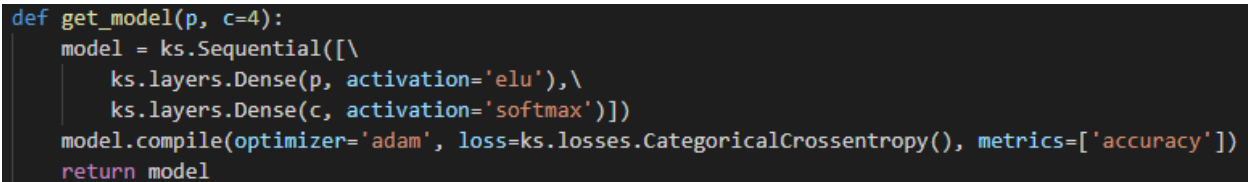
https://github.com/EAGrimaldi/EECE5644_Assignment_3

Appendix B: Figures



Empirical results of theoretically optimal classifier
Success rate: 0.848760
Failure rate: 0.151240

Figure 1: results of theoretically optimal classifier on 100000 samples



```
def get_model(p, c=4):  
    model = ks.Sequential([\br/>        ks.layers.Dense(p, activation='elu'),\  
        ks.layers.Dense(c, activation='softmax')])  
    model.compile(optimizer='adam', loss=ks.losses.CategoricalCrossentropy(), metrics=['accuracy'])  
    return model
```

Figure 2: implementation of an MLP with a hidden ELU layer and a Softmax Output layer

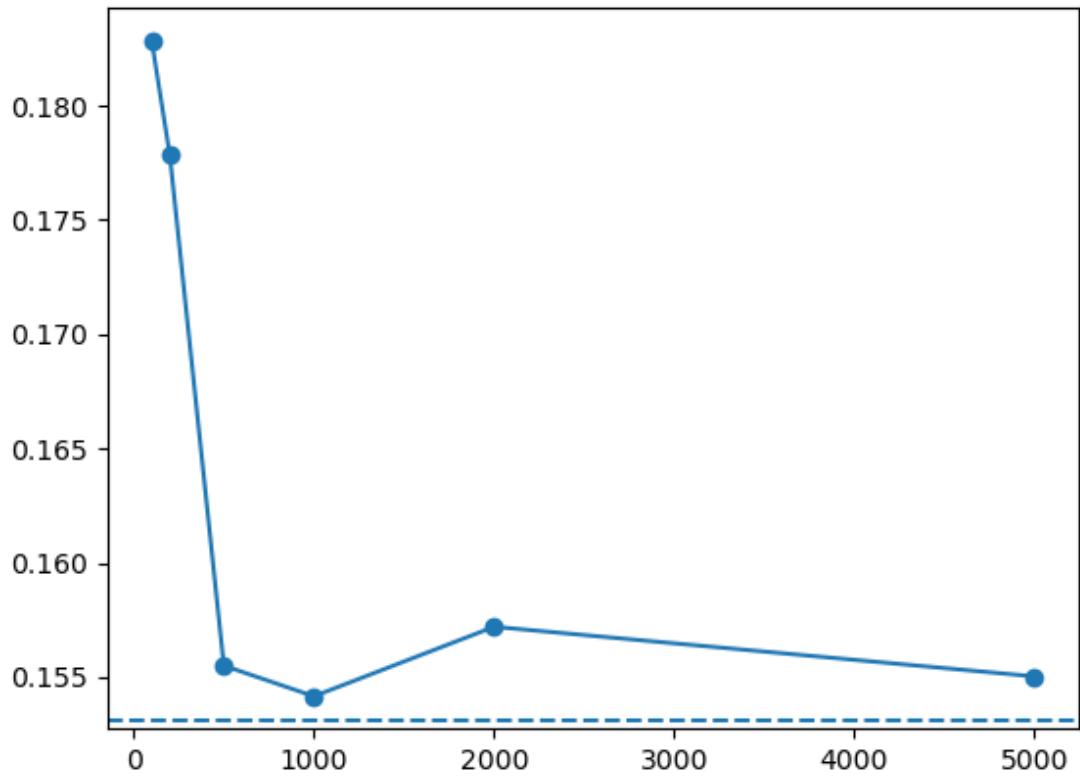


Figure 3: samples vs empirical \hat{P}_e for the MLP; the dotted line is the theoretical \hat{P}_e

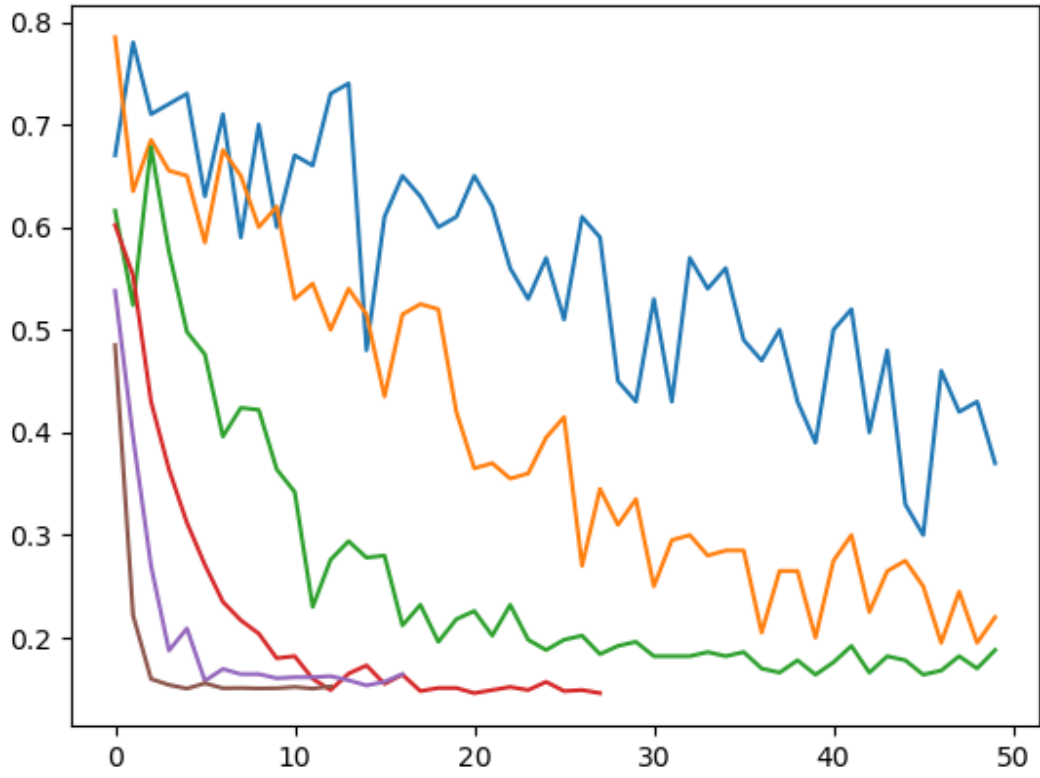


Figure 4: perceptrons vs empirical \hat{P}_e for the MLP
N=5000 is brown
N=2000 is purple
N=1000 is red
N=500 is green
N=200 is yellow
N=100 is blue

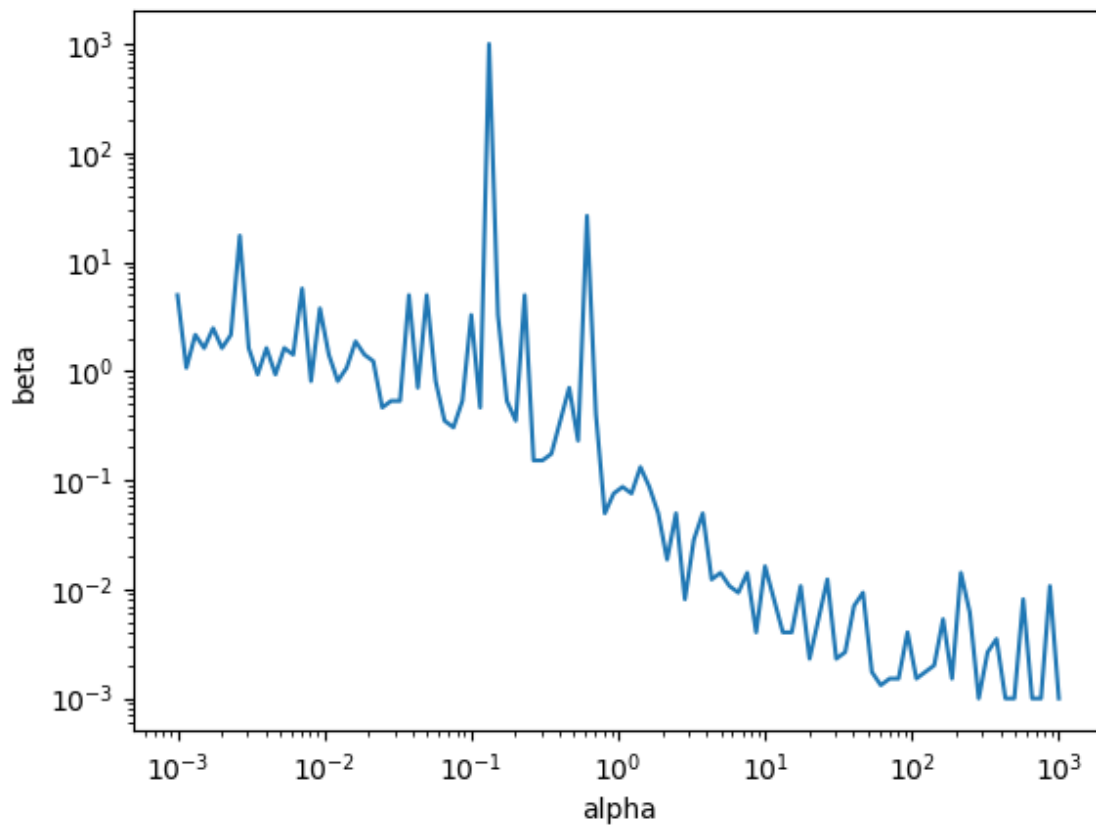


Figure 5: alpha vs beta

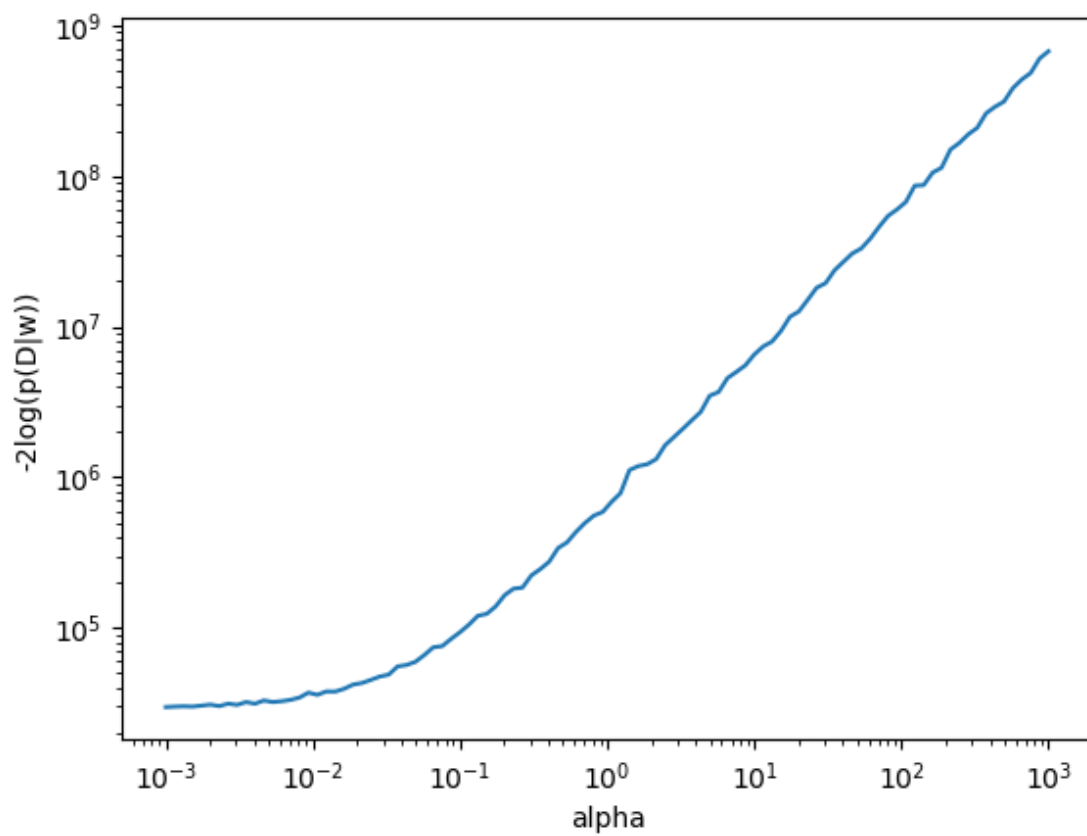


Figure 6: α vs $-2\ln p(D|w)$

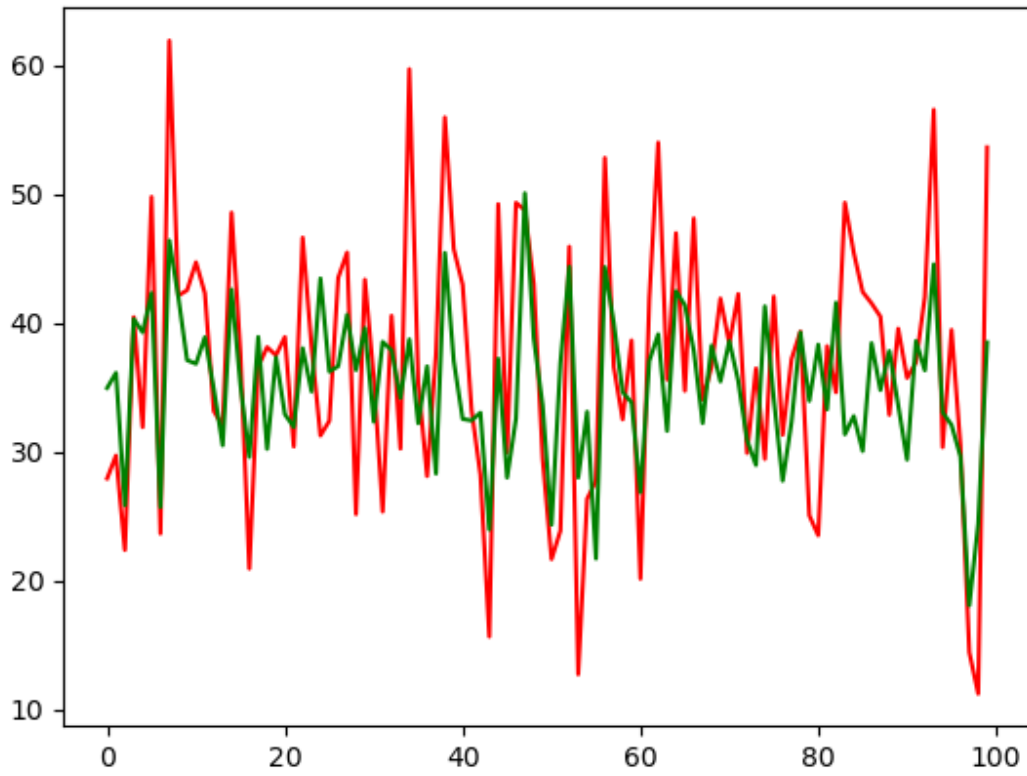


Figure 7: the true and estimated values of y for the first 100 samples of D_{test}
this example uses $\alpha = \frac{trace\Sigma}{7}$ to demonstrate "kind of" tracking
 y_{true} is red
 y_{est} is green