# Hand in 1: OCR and Logistic Regression

Here in this problem, we are not going to define the concepts of logistic regression, negative log likelihood, and also gradient descent algorithm; but instead we will implement the gradient descent algorithm and discuss the parameter we are going to use in it, and see how manipulating these parameters can affect the performance of our model.

## Binary classification (2s vs 7s)

At the first part of the problem, we must separate the twos and sevens, so we confront with a binary classification problem. Since the output of logistic regression has a value between 0 and 1, it can be a good candidate for separating the twos and sevens i.e. the digit (image) is a 2 or not. For being able to use the logistic regression algorithm as a binary classifier, we must define a threshold variable that means the outputs below it would be considered as zeros (so the digit is two), and the values more than it would be considered to ones (so the digit is seven).

### Negative log likelihood (NLL)

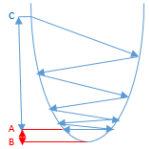
Likelihood means the probability of having the expectation output (after data or inputs are available). So in our problem (being 2 or not being 2), we are going to maximize the function that calculates the output (here 0 or 1 due to have a logistic regression) given the inputs (images). Log-likelihood is the natural logarithm of likelihood that working with it is more convenient. Maximizing the likelihood estimation means minimizing the negative log likelihood, and we will use gradient descent algorithm for minimizing it.

### Gradient descent algorithm

***Gradient descent algorithm*** uses a ***step size*** parameter (***learning rate***) to gradually move to the minimum value of our ***convex*** function (here ***NLL***, that the assumption of being ***convex*** and also particular values of ***step size***, guarantee the movement to the minimum value). Actually, in the algorithm, we start with an initial solution (weights) and take the gradient at that point. The solution is moving into the negative direction of the gradient in size of ***step size***, and again take the gradient at the new point.

We repeat the process until the change in the value of NLL reaches its minimum value – we put a condition for the difference between the new and the old value of NLL. So, we can be sure that repetition of the algorithm cannot result in smaller value for NLL and we have reached the minimum local point.

The only remaining point here is selecting the step size. Selecting larger amounts for step size can result in moving faster into the minimum point (of the cost function – here NLL), but at the same time we may miss reaching the smallest point we can get it; because a large step size may lack the power of jump at the exact minimum point – while the difference between old and new calculated cost functions touches the exit condition.




The difference of old and new cost function reaches the threshold (the condition of exiting the loop), but it cannot reach the minimum local point of the function. The solution can be decreasing the value of stop condition.

Very small values for learning rate however can touch the minimum local point of the cost function, but they cause very small changes in each iteration for it, and therefore very slow movement to it. Another problem is the difference between old and new values of the cost function is not too much in this case, and therefore we also must decrease the threshold for stop condition of the loop, to make sure that ultimately the movements can reach the minimum point.

In each repetition, we can change the value of step size, but here we are not going to do that due to decreasing the complicacy of implementation. We select 0.005 as a default value for $\eta$ (step size) at the first part of the problem (2s vs 7s) and use the stop condition of 0.005. Finally, we select these two numbers for our full-batch-gradient-descent model.

| Step size | 0.01 | ***0.01*** | ***0.005*** | 0.001 | ***0.001*** |
|---|---|---|---|---|---|
| Misclassification rate | 3.88 | 3.68 | ***3.68*** | 11.05 | 3.68 |
| Stop criterion | 0.005 | 0.001 | ***0.005*** | 0.005 | 0.001 |
| Number of repetition | 30 | ***86*** | ***33*** | 1 | ***164*** |

## *Mini-batch stochastic gradient descent*

Besides calculating the gradient using all the data, this time we select a limited number of random records and repeat the calculation of the gradient for each of these training examples (mini-batch). This can help to have better performance in time due to having a smaller collection of records – ***batch size***. But because we want to have a better performance also in misclassification rate, we repeat the process more than one time - ***epoch***. We do not expect that this kind of calculation of gradient works as well as when we use all training sample for it, but at least it can perform quiet well (especially in situations with much higher number of records – training samples). Because stochastic selection of the records, we see different misclassification rates for different running of the mini-batch function. We expect higher number of ***batch sizes*** and ***epochs*** result in better misclassification rates. But while executing the program we noticed some of higher epochs or batch sizes takes less time to execute. So, we decided to plot the cost function as a function of repetition number.

## *Samples of misclassification*

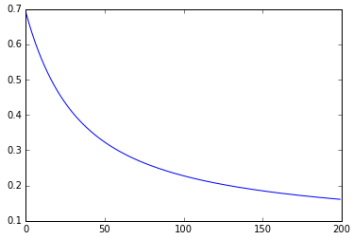
First 16 mistakes from full-batch-gradient-descent are:

From the mathematics perspective, the reason of misclassification for these numbers is the points in images that have close values for both what we have learned from the training samples and what we see now in test sample. The model has arranged its weights based on the values of each 784 data points – that is a number between 0 and 255. For example, it has been learned that pixel $(a, b) - 0 \leq a, b < 28$ – in sevens has a value close to $c$. This means we expect if we see the value of $c$ for that corresponding pixel in a new image, the number must be 7. So, more interference (having the same values in all three dimensions for two different number) results in more misclassification rates. 2s and 7s in writing are very similar in that measure, so we expect to have interference in values of corresponding pixels, although I think our model performs really nice.

From the imagination perspective, if we consider a 2 with a curve in top, a diagonal line from top to bottom, and finally a horizontal line in bottom. In some cases of misclassification collection, we can see the bottom line of 2s has little distance from the top, so model recognizes this line as the middle line of seven - we think it is because the most training 7s have been had a middle horizontal line. Also, we can see the sevens with a curve line at the top are among the mistakes, because the model has learned to set the weights ($\omega$) in order to recognize the curve line at the top as a 2. Furthermore, we see some cases that the middle line of 7 has a long distance from the top, and it is closer to the bottom than to the top, so model recognizes them as the bottom line of a 2.

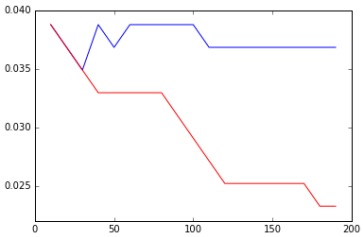
## *Cost vs repetition*

We have plotted the value of cost function (NLL) based on the number of repetition in full-batch-gradient-descent model – although we had decided to use the stop criterion for our full and mini batch-gradient-descent, the performance of repetition (in comparison with stop criterion) was better especially in OCR.



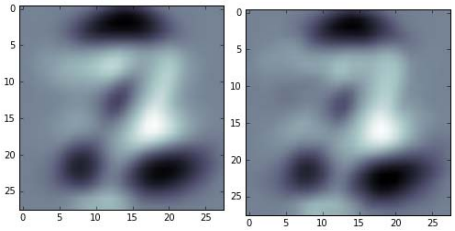
## *Performance vs time*

While both of full and mini batch algorithms would be decreasing when the number of iterations increases, the full-batch (blue) here – for binary classifier - does not change very much; however, the mini-batch decreases.



## *Parameter vectors*

You can see the plot of parameter vectors of the full batch with stop criterion = 0.005 and step size = 0.005 at the left side, the plot of the mini batch with batch-size = 1000 and epochs = 500 has plotted on the right side.



# *OCR*

First of all, for each digit, we change the labels of our training sample to 0s and 1s for that digit, and run the full batch algorithm to get parameter vector for that digit. So, we have 10 parameter vectors. Then we run each of these 10 on our test sample to calculate the predicted values. This time the predicted values are not just 0s and 1s (because we need the log values (between 0 and 1) to be able to compare them. Each vector that makes the least log value for the sample data point would be selected. Then we compare our predicted values with the test labels.

Result is %14.96 misclassification rate in 1000 times repetition in full batch algorithm. The parameter vectors are:

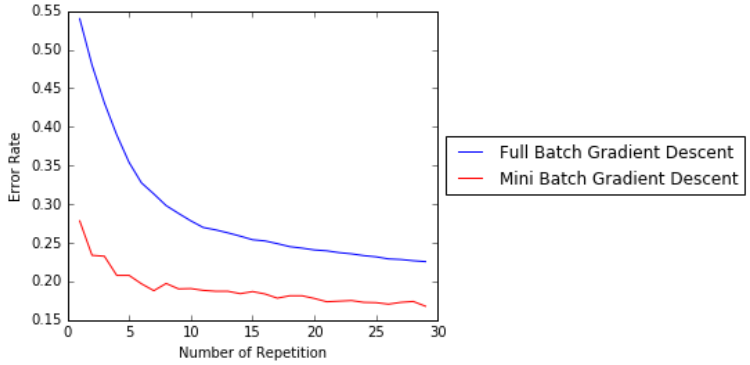
Below are the first 16 mistakes. Our prediction:

***[8 7 6 5 9 0 0 3 5 8 8 3 1 4 8 8]***

While the numbers were:

***[3 2 2 9 8 6 8 5 6 6 2 8 9 1 3 4]***

We used time for our measure but the form of plot was not very well (too much fluctuations). So again, we are going to use repetition and consider the time of each execution of mini-batch approximately as 1/10 of that of full-batch – the whole records are 10380 and batch-size is 1000. Also, I have compared the error rate for two implementations (mini = red, full = blue).




## Sanity check

If we permute the points (pixels), we think the classifier would be the same, because the place of the points does not matter and we can reshape the $28 \times 28$ matrix into any other shape or array. What we learn from it is the values of each pixel for different numbers (digits), so the main point in comparison between two numbers is the shared pixels. In other words, more pixels in the same place that carry out the general pictures of the two numbers, results in more misclassification rate. For example, while misclassification between 2 and 7 are %3.68, the rate between 3 and 4 is %1.94.

## Regularization

Actually, the regularization term would be added to the gradient function to inhibit the model from overfitting. Because we do not have to use the term with bias parameter, we change the batch and mini gradient functions. The result is being improved from %3.88 misclassification rate to %3.68 for regularization parameter of 10000 – in 100 repetitions. But in some cases, the parameter has a vice effect on the performance e.g. from %3.29 ($\lambda = 0$) to %3.49 ($\lambda = 5000$) for 5000 repetitions.

## Linear separable

In a binary classification, using implementation of gradient descent for logistic regression, higher numbers of repetition cause the weights to converge to specific values. In case the data is linear separable, these weights tend to be at their extreme states (0 or 1). This makes the figure of vector parameters (weights) clearer to show the two numbers. But in the practice, it is not going to happen because we have some pixels that are shared between two numbers. And having more share pixels means less power to separate the data in a linear way.

In too many repetitions, adding the regularization cannot change the fixed numbers (0s or 1s) that the weights are converging to. It just adds a noise to the model, so we can have less bias (underfitting) and more powerful model for prediction.

## Bonus Question

As we said, regularization parameter just adds a noise to the NLL function, but the whole shape of it does not change. Of course, high values for the regularization parameter i.e. more variance can affect the convexity of the NLL function and therefore moving in negative direction of gradient may not result in decreasing the gradient - as we saw in our binary classification problem (5000 repetitions) – and therefore it can worsen the performance.

## Multinomial/Softmax Regression

The algorithm is the same with logistic regression while just uses the ***softmax*** function instead of logistic function – and therefore it is used for classifying more than two categories at the same time. The implementation has just two points. First, for each data point (digit) the prediction function returns a number between 0 and 1 – and not 0 or 1 as we had in logistic algorithm - for each category that shows the probability of being in that category. Second, because of summing up the exponential terms, we would have a very large number in the denominator of the ***softmax*** term.

So, we can subtract the max of exponentials to have a much less denominator without changing the comparative probabilities. Both implementations have been brought into the code.

## *Parameter vectors*

(1000 full-batch repetition – %14.22)

## *Mistakes*

The same with logistic (page 4).

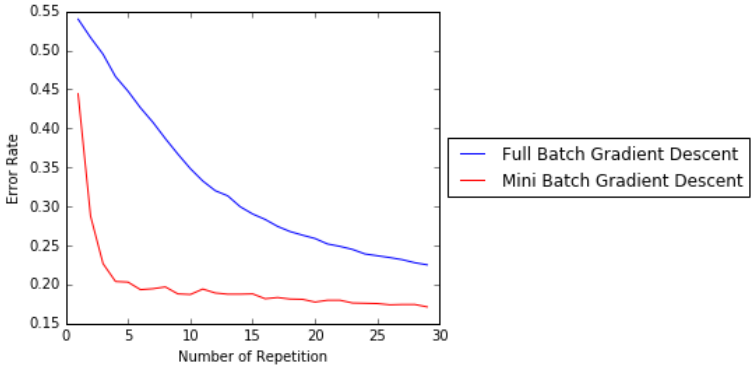
## *Results (Error rates)*

(with 100 repetitions – mini-batch algorithm with batch-size of 1000)

| Algorithm | AU | Mnist |
|---|---|---|
| Full | 18.91 | 24.18 |
| Mini | 19.11 | 24.14 |

## *Performance vs time*

We used time for our measure but the form of plot was not very well (too much fluctuations). So again, we are going to use repetition and consider the time of each execution of mini-batch as 1/10 of that of full-batch – the whole records are 10380 and batch-size is 1000. Also, I have compared the error rate for two implementations (mini = red, full = blue).




Totally, the power of mini-batch in terms of time of executing is better than full-batch.

# Hand in 2: OCR with SVM and (Deep) Neural Nets

## SVM

Because we are told that we must use the validation set as a part of training set, we will measure the generalization performance of the models with cross-validation (on the training set). ***Hyperparameters*** are parameters that are not learned directly from the estimators, and in scikit-learn we are going to pass them to the constructor of the estimator classes. It is possible and recommended to search the hyper-parameter space for the best Cross-validation: evaluating estimator performance score.
Any parameter provided when constructing an estimator may be optimized in this manner. Specifically, to find the names ...

### Linear Kernel

In linear kernel, $C$ parameter (cost for misclassification) identifies how much our optimization (using ***linear kernel***) - ***linear hyperplane***, or ***linear boundary decision*** is sensitive about outliers (It is when our data is linear separable). Higher values of $C$ means we assume higher cost for outliers and therefore the SVM optimization tries to select a smaller margin hyperplane, so fewer the number of misclassified points. At the same time, because overfitting with training data, the generalization power of the model decreases (lower bias and higher variance).
At the same way, if we choose the $C$ parameter smaller, the hyperplane will have larger margin for its separator, and also better performance on new data (higher bias and lower variance). But it is not always the case, and it is very dependent on the data itself. So we have to gradually increase (or decrease) the $C$ to see when it starts to overfitting, and see how it affects the performance of our model.
Here (and also usually) a power change value is considered for $C$, because it can show the performance clearer. We have used linear kernel with different values of $C$ and after looking into the shapes of digits (here zero as a sample) we can conclude the image of digit is more clear, when $C$ is larger. It is because using smaller margins results in more fitting with the training data.

### Polynomial Kernel

Polynomial Kernel is a kernel function that represents the similar vectors in feature space (training samples), but this time it also looks in feature spaces of polynomials of original values that makes it enable to learn non-linear models. In polynomial $K(x.y) = (x^T y + c)^d$ we have two hyperparameters: degree of polynomial $(d)$ and $c \geq 0$ that is a free parameter trading off the influence of higher-order versus lower-order terms in the polynomial.
Actually it means we have a bold distinction problem with our pairs of data $(x.y)$. When the degree of $d$ increases, the distinction between pairs which are less than $1$ and the pairs more than $1$ becomes more and more. To overcome this problem, we are going to set $c = 1$ to compensate the non-symmetric data. Besides it we are using the $c = 0$ (homogeneous).

### RBF Kernel

RBF Kernel means we have a soft separator for our points. Although it is designed for separating (or actually clustering the points (vectors) without labels, we expect RBF performs better than two other models.
With the kernel $K(x.y) = \exp(-\gamma \left|\left|x_i - x_j\right|\right|^2)$ we can change the kernel and here two parameters are important: $C$ and $\gamma$. $C$ is the same with linear kernel. A small $\gamma$ means a Gaussian with a large variance so the influence of $x_j$ is more, i.e. if $x_j$ is a support vector, a small $\gamma$ implies the class of this support vector will have influence on deciding the class of the vector $x_i$ even if the distance between them is large. If $\gamma$ is large, then variance is small implying the support vector does not have wide-spread influence. Technically speaking, large $\gamma$ leads to high bias and low variance models, and vice-versa. So we are going to use some values for $\gamma$ and interpret the result.
We have used the polynomial and rbf kernels for our model, we have assumed that maybe our problem is not linear separable. When we are using rbf, $\gamma$ controls the shape of the ***peaks***. A small $\gamma$ gives us a pointed bump in the higher

dimensions, a large $\gamma$ gives us a softer, broader bump. So a small $\gamma$ will give us low bias and high variance and vice versa.
The effectiveness of SVM in non-linear separation depends on the selection of kernel, the kernel's parameters, and soft margin parameter $C$.

## Grid Search

We select the best combination of hyperparameters for our classifiers using grid search, that means for each classifier we test all values of hyperparameters - that grow usually with exponentially sequences of $C$ and $\gamma$.

## Cross Validation

We are told that ***the validation set should come from the training data not the test data***, and this is the meaning of using cross-validation that is used by grid search. Actually, each combination of parameter choices is checked using cross validation, and the parameters with best cross-validation accuracy are picked. However, at the end we are checking the best parameters on the validation set (%30 of our training set)

```
Best parameters set found on development set:

{'gamma': 0.001, 'C': 100, 'kernel': 'rbf'}

Grid scores on development set:

0.908 (+/-0.011) for {'gamma': 0.001, 'C': 1, 'kernel': 'rbf'}
0.860 (+/-0.014) for {'gamma': 0.0001, 'C': 1, 'kernel': 'rbf'}
0.936 (+/-0.007) for {'gamma': 0.001, 'C': 10, 'kernel': 'rbf'}
0.905 (+/-0.009) for {'gamma': 0.0001, 'C': 10, 'kernel': 'rbf'}
0.939 (+/-0.005) for {'gamma': 0.001, 'C': 100, 'kernel': 'rbf'}
0.927 (+/-0.008) for {'gamma': 0.0001, 'C': 100, 'kernel': 'rbf'}
0.939 (+/-0.003) for {'gamma': 0.001, 'C': 1000, 'kernel': 'rbf'}
0.920 (+/-0.006) for {'gamma': 0.0001, 'C': 1000, 'kernel': 'rbf'}
```

With more values of $\gamma$ we see the effect of distance among the data points more in our model. apparently, with the same C, lower values of $\gamma$ cause the lower performance. It is because with lower values of $\gamma$ we are more sensitive about the distances between data points and also adding a new data points makes more changes in our separator (in terms of place of it); and it means higher variance and lower bias. But exactly because this property, they may perform better on test samples.
With higher values for $C$ again – less values for margin, the adaptation power of model increases – so the accuracy rate in training sample, but it may cause the overfitting – and performs worse on test data.

```
0.909 (+/-0.011) for {'C': 1, 'kernel': 'linear'}
0.907 (+/-0.014) for {'C': 10, 'kernel': 'linear'}
0.907 (+/-0.014) for {'C': 100, 'kernel': 'linear'}
0.907 (+/-0.014) for {'C': 1000, 'kernel': 'linear'}
```

When we are increasing the $C$, we are considering smaller margins for our model that results in better performance on training sample. However, due to limitations the linear model has, there is no much difference among performance in linear models.

```
0.901 (+/-0.012) for {'degree': 1, 'coef0': 0, 'kernel': 'poly'}
0.884 (+/-0.014) for {'degree': 2, 'coef0': 0, 'kernel': 'poly'}
0.827 (+/-0.009) for {'degree': 3, 'coef0': 0, 'kernel': 'poly'}
0.738 (+/-0.013) for {'degree': 4, 'coef0': 0, 'kernel': 'poly'}
0.901 (+/-0.012) for {'degree': 1, 'coef0': 1, 'kernel': 'poly'}
0.912 (+/-0.009) for {'degree': 2, 'coef0': 1, 'kernel': 'poly'}
0.923 (+/-0.007) for {'degree': 3, 'coef0': 1, 'kernel': 'poly'}
0.932 (+/-0.009) for {'degree': 4, 'coef0': 1, 'kernel': 'poly'}
```

As we can see, when the degree of polynomial increases, $c$ values of 1 (the parameter for controlling the distance between pairs less than 1 and pairs more than 1) plays its main role to overcome the non-symmetric problem., while with c = 0, we can see the performance is worse.

***Detailed classification report:***

***The model is trained on the full development set.***
***The scores are computed on the full evaluation set.***

| | precision | recall | f1-score | support |
|---|---|---|---|---|
| 0 | 0.99 | 0.99 | 0.99 | 298 |
| 1 | 0.91 | 0.95 | 0.93 | 334 |
| 2 | 0.96 | 0.99 | 0.97 | 307 |
| 3 | 0.95 | 0.97 | 0.96 | 304 |
| 4 | 0.92 | 0.90 | 0.91 | 294 |
| 5 | 0.95 | 0.95 | 0.95 | 294 |
| 6 | 0.95 | 0.99 | 0.97 | 319 |
| 7 | 0.96 | 0.95 | 0.96 | 314 |
| 8 | 0.97 | 0.89 | 0.93 | 324 |
| 9 | 0.91 | 0.90 | 0.90 | 326 |
| avg / total | 0.95 | 0.95 | 0.95 | 3114 |

we have same result for both ways of scoring (precision and recall), so we will not repeat the results from recall.

### *Best SVM on Test sample (Results)*

| | precision | recall | f1-score | support |
|---|---|---|---|---|
| 0 | 0.96 | 0.99 | 0.98 | 258 |
| 1 | 0.92 | 0.94 | 0.93 | 258 |
| 2 | 0.96 | 0.97 | 0.97 | 258 |
| 3 | 0.94 | 0.95 | 0.95 | 258 |
| 4 | 0.97 | 0.94 | 0.96 | 258 |
| 5 | 0.96 | 0.91 | 0.94 | 258 |
| 6 | 0.98 | 0.97 | 0.97 | 258 |
| 7 | 0.95 | 0.96 | 0.96 | 258 |
| 8 | 0.94 | 0.92 | 0.93 | 258 |
| 9 | 0.90 | 0.93 | 0.92 | 258 |
| avg / total | 0.95 | 0.95 | 0.95 | 2580 |

The results are the same with results in validation sample.

## *Neural Nets*

We are going to show a model with three layers: input, one hidden layer, and output. We proceed with the parameters that can affect the performance of the model: the number of layers (fixed here), the number of units in each layer (we try it later), and the activation function for hidden layer and output layer. We use ***Softmax Regression*** ($y = softmax(evidence)$) as the activation function for our output layer, which ***evidence*** is a linear function ($\text{evidence}_i = \sum_j \text{W}_{i,j} x_j + b_j$) that we replace it with a ***relu*** function as the activation for the hidden layer.

### *Defining variables, activation, and loss function*

In ***tensorflow*** we define the input variables (images and labels) as two tensors (placeholders). Also because we are using the linear relationships between all samples from input layer at the hidden layer (actually ***relu***), we have to define two variables (W and b). The questions in this step are:

1. How many nodes we must have at the hidden layer?
2. How we are going to change the weights and biases (the algorithm for finding the optimum)?
3. What is our validation sample to improve the performance of the model?

Actually in the problem, we have 784 ($28 \times 28$) nodes in the input layer, and 10 (each for a digit) nodes in the output layer. We will try different number of nodes in the hidden between this two number. Furthermore, we are going to

use the ***Gradient Descent*** algorithm for minimizing the loss function. Also we are going to use ***cross-entropy*** algorithm for out ***training*** step. After that we ***test*** the performance of the model on the test sample.

## *Training*

After defining the structures (tensors, placeholders, variables), and adding operations; our ***Neural Network*** using data samples must learn how to change the hyperparameters to have them the best for our classification problem. We wanted to read a random subsample of our training data in a loop, so we could check the optimum number of repetition of the loop (tradeoff again between variance and bias); but because the ***images*** and ***labels*** are in order, so we just repeat selecting a subset n times and move thorough all records we have in training sample - actually I cannot check the performance of reading samples for my subsamples in a random way.
We started with a 10 ***hidden units*** and then try with other numbers:

Training vs. Test accuracy rates (10 hidden unit)

| | **Repetition No.** | | | | | |
|---|---|---|---|---|---|---|
| **Batch Size** | 10 | | 100 | | 500 | |
| 100 | 0.909056 | 0.893411 | 0.941715 | 0.906202 | 0.956647 | 0.905814 |
| 1000 | 0.869557 | 0.866279 | 0.912813 | 0.895349 | 0.937958 | **0.906589** |
| 10000 | 0.69711 | **0.695349** | 0.874952 | 0.872093 | 0.899807 | 0.886047 |

As we see, the ***repetition numbers*** more than 100, causes overfitting, because we can see by changing the repetition number from 100 to 500, the learning power of our model regarding the training sample slightly change and also the classification power of it on test sample, decreases.
Also we can see with larger amounts of ***Batch Size***, overfitting decreases and the classification power of model increases. It happens until the batch size of 1000. After that increasing the batch size, causes the prediction power decreases. Furthermore, we see when the batch size is small, the distance between accuracy rate of training and test samples grows. It is because the model with smaller batch sizes has more opportunity to learn data and its delicacy.
The last point about the accuracy rates above is when we have just 10 unit nodes in the hidden layer, the accuracy rates of linear activation function (hidden) is slightly better than it of ***relu*** – and I have used them for above table.

Training vs. Test accuracy rates (500 hidden unit)

| | **Repetition No.** | | | |
|---|---|---|---|---|
| **Batch Size** | 100 | | 500 | |
| 100 | 0.986898 | 0.951938 | 0.990751 | 0.957364 |
| 1000 | 0.94499 | 0.926744 | | |

We can conclude with higher number of unit nodes in the hidden layer, the power of learning any behavior of the data grows up. Also the difference between accuracy rate of training sample and test sample increases.

# *Competition*

I wanted to use the Neural Network model, but I have a windows laptop and tensorflow just runs in VM. So I cannot useful functions of sklearn (for manipulating the images and therefore generating more samples). Furthermore, I implemented the Tensorflow Convolutional Nets, but in my laptop it was very slow. I have attached my ipynb file and I will proceed with the SVM.

## *Changing in images*

I have used cropping method (and then reshape), flipud, and rotate for the images. It makes the number of training samples four times more. Also I have used the ***Dimensionality Reduction*** using ***IncrementalPCA***. I have attached the train.py and predict.py.

# Hand in 3: A hidden Markov model for gene finding in prokaryotes

Before anything we assume that every observed variable has included three single elements – in C and R states. After that we divide up the status of being in each of C or R states, into three separate states: Start, Stay, and Stop. And also we assume that when we enter in the start and stop states, we do not stay in that states.

## Deciding on initial structure

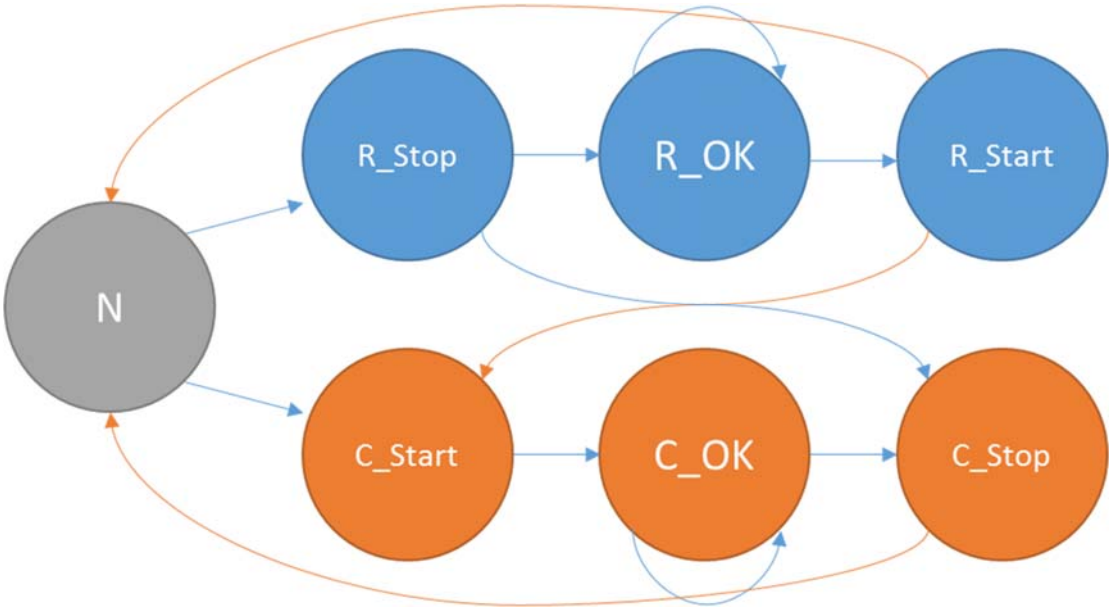
The main point is which states are important for changing from three hidden states (N, R, and C) to each other. So we have to consider the states (including three elements), for starting and ending the Rs and Cs. So we have four observed variables (categories). To see which three-elements observed variables are in each of these four categories, we had to learn from real data - first five annotations and genomes. The best news is the elements we see in each of these states are different. The bad news is we see all the collections of three elements in Ns, Cs, and Rs.

### Hidden variables

So we have seven states for our hidden variable: C_Start, C_OK, C_Stop, R_Stop (because the direction of Rs is vice versa from the left to right in comparison with Cs, and we see R_Stop first), R_OK, R_Start, N - $K = 7$.

### Observed variables

C_Start: GTG, GTT, ATC, TTG, ATA, ATT, ATG. R_Start: CAC, TAT, AAT, CAG, CAT, GAT, CAA. R_Stop: TCA, TTA, CTA. C_Stop: TAA, TAG, TGA. So we have only four categories of observed variables - $D = 4$.




## Tune model parameters by training

We use training by counting technique. Below are the tables for $A$, $\varphi$, $\pi$ that we have learned from the first five files – in order of RST, ROK, RSR, CSR, COK, CST, NOK for both columns and rows. A:

```
[[  0.00000000e+00   1.00000000e+00   0.00000000e+00   0.00000000e+00
    0.00000000e+00   0.00000000e+00   0.00000000e+00]
 [  0.00000000e+00   9.91233478e-01   8.76652245e-03   0.00000000e+00
    0.00000000e+00   0.00000000e+00   0.00000000e+00]
 [  6.30887185e-02   0.00000000e+00   0.00000000e+00   2.19058050e-04
    0.00000000e+00   0.00000000e+00   9.36692223e-01]
```

```
[  0.00000000e+00   0.00000000e+00   0.00000000e+00   0.00000000e+00
   1.00000000e+00   0.00000000e+00   0.00000000e+00]
[  0.00000000e+00   0.00000000e+00   0.00000000e+00   0.00000000e+00
   9.91392271e-01   8.60772884e-03   0.00000000e+00]
[  2.94674805e-03   0.00000000e+00   0.00000000e+00   7.05114713e-02
   0.00000000e+00   0.00000000e+00   9.26541781e-01]
[  1.03713543e-02   0.00000000e+00   0.00000000e+00   1.07435845e-02
   0.00000000e+00   0.00000000e+00   9.78885061e-01]]
```

And $\varphi$ with order or RST, RSR, CSR, CST for columns:

```
[[  1.00000000e+00   0.00000000e+00   0.00000000e+00   0.00000000e+00]
 [  8.25762246e-05   3.45902203e-01   4.69472722e-01   1.84542499e-01]
 [  0.00000000e+00   1.00000000e+00   0.00000000e+00   0.00000000e+00]
 [  0.00000000e+00   0.00000000e+00   1.00000000e+00   0.00000000e+00]
 [  1.75300963e-01   4.30480634e-01   3.94155004e-01   6.33986762e-05]
 [  0.00000000e+00   0.00000000e+00   0.00000000e+00   1.00000000e+00]
 [  1.74948910e-01   3.15777053e-01   3.33400642e-01   1.75873394e-01]]
```

And we have all elements for $\pi$ as zeros but the element of N, because always the string starts with a N.

## Reading the observations

In the first step, we learn from five first annotations for our genomes and then generate the annotation strings for each of second five genome files.

After that to calculate the performance of our model using only the first five annotations, we apply a cross-validation algorithm to how our algorithm (first learning from four other files, and then using Viterbi to calculate the performance) works.

The ACs for first five genomes are:

| Genome | C | R | Both |
|---|---|---|---|
| 1 | 0.3016 | 0.4339 | 0.1151 |
| 2 | 0.3526 | 0.3426 | 0.0613 |
| 3 | 0.3811 | 0.3440 | 0.1078 |
| 4 | 0.3792 | 0.3734 | 0.1346 |
| 5 | 0.3492 | 0.3145 | 0.0492 |

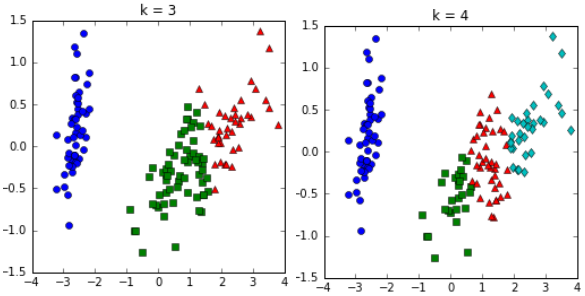
## Improving the algorithm

We have not separated the three-elements for each observed state, and calculated them in a same observed state. For example, we have considered all states of TCA, TTA, and CTA in the R_Stop step. But we could also see these elements as a separate observed state, and therefore we could have 20 separate observed states. I think it could be useful in terms of performance of the model.

# Hand in 4: Implementing and using representative-based clustering algorithms

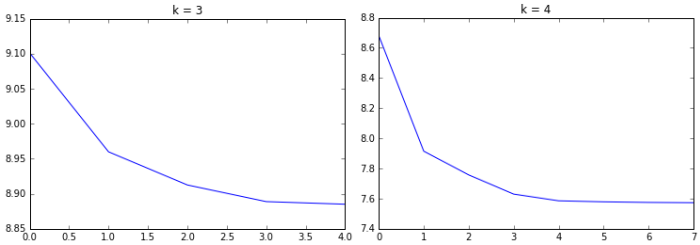
## Implementing K-mean algorithm

Using the $k = 3$ as a default results in some misclassifications – regarding the labels the data already has. The reason is the ***K-mean*** algorithm assumes that the data has a convex shape, and does not consider concepts such as density and connectivity – among data points. So we see in the figures, the only measure here is the distance from just a center – maybe we would have a better performance if we had more than a center for each of our clusters, but I do not know about the existing algorithms. The figures below use the four-dimensional data of ***Iris***.




## Cost function for the K-means problem

To evaluate the performance of our model, we can use the ***Total Distance*** of a cluster, as the cost function (also ***compactness of a clustering*** as ultimate objective function):



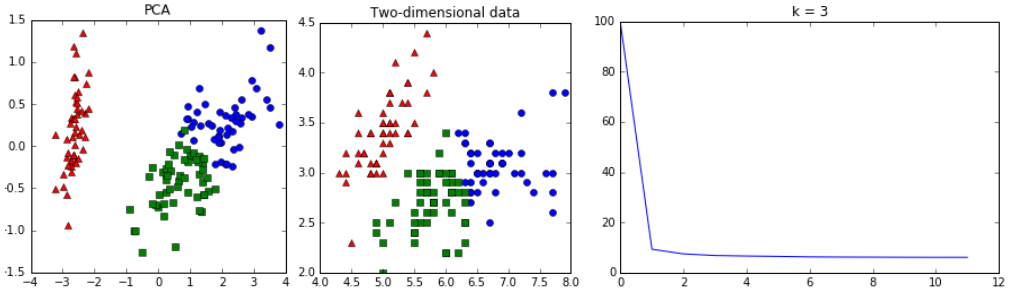

The above figures show the compactness of clustering based on the number of iteration of the algorithm – until the distance among old centers and new centers is less than a very small number ($\varepsilon$). So the cost function decreases by increasing the number of iterations.

## Comparison with the sample from the book

First of all, I have to say I have run the algorithm for just two dimensions of the data. First, I ran it on the first two dimensions (from four) of the data. After using the centers from the book: (I have assumed the $k = 3$):

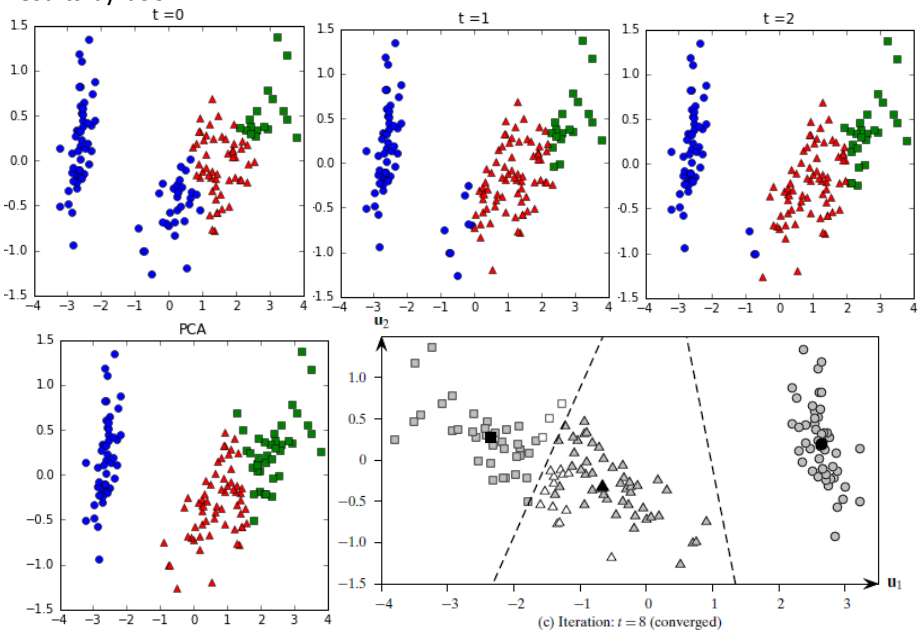
$$\mu_1 = (-0.98, -1.24)^T \qquad \mu_2 = (-2.96, 1.16)^T \qquad \mu_3 = (-1.69, -0.80)^T$$

Here the problem was I encountered with two empty collections for second and third clusters. I added a random point to them and continued. Also for two-dimensional data (just the first two columns) I have run the algorithm, and I have plotted both of PCA and two-dimensional data:




At the PCA figure, we see some intervals among data points exists, and it is because of using other dimensions rather than PCA two-dimensional data points.

At the next step, I have used the two-dimensional data points of PCA. The point is the direction of x-axis in the book is opposite of our data, so I change the first values (x values) of three centers multiplied by -1, and then I have ended up with the same results by book:




Furthermore, our algorithm is similar to the book in terms of the number of repetition of algorithm. The final means from the book:

$$\mu_1 = (2.64, 0.19)^T \qquad \mu_2 = (-2.35, 0.27)^T \qquad \mu_3 = (-0.66, -0.33)^T$$

And the results of our algorithm:

```
array([[-2.64084076, 0.19051995], [2.34645113, 0.27235455], [0.66443351, -0.33029221]])
```

# Implementing EM algorithm

I have used the three points of the book and also the three points of K-mean implementation as our random means for EM algorithm.

## Comparison with the sample from the book

If we compare the results of our algorithm with that from the book ($\epsilon = 0.001$ $and$ $k = 3$), we see our algorithm has approximately the same results in mean points, but at the same time I think our algorithm performs better in term of ***number of iterations*** (we: 25, book: 36). The final mean points from the book for the dependence covariance matrix are:

$$\mu_1 = (-2.02, 0.017)^T \qquad \mu_2 = (-0.51, -0.23)^T \qquad \mu_3 = (2.64, 0.19)^T$$

$$\Sigma_1 = \begin{pmatrix} 0.56 & -0.29 \\ -0.29 & 0.23 \end{pmatrix} \qquad \Sigma_2 = \begin{pmatrix} 0.36 & -0.22 \\ -0.22 & 0.19 \end{pmatrix} \qquad \Sigma_3 = \begin{pmatrix} 0.05 & -0.06 \\ -0.06 & 0.21 \end{pmatrix}$$

$$P(C_1) = 0.36 \qquad P(C_2) = 0.31 \qquad P(C_3) = 0.33$$

And our results are:

```
mean: [[ 2.0290574  0.01841914], [ 0.51302185 -0.22478236], [-2.64084076  0.19051995]]
cov: [[[ 0.55878386 0.29276142], [ 0.29276142  0.23260583]]
      [[ 0.36551596 0.21756254], [ 0.21756254  0.18773886]]
      [[ 0.04777048 0.05590782], [ 0.05590782  0.21472356]]]
prior: [ 0.33495113  0.33150901  0.33333333]
```

Also if we run our algorithm by diagonal covariance matrix assumption, the book results (29 iterations) are:

$$\mu_1 = (-2.1, 0.28)^T \qquad \mu_2 = (-0.67, -0.40)^T \qquad \mu_3 = (2.64, 0.19)^T$$

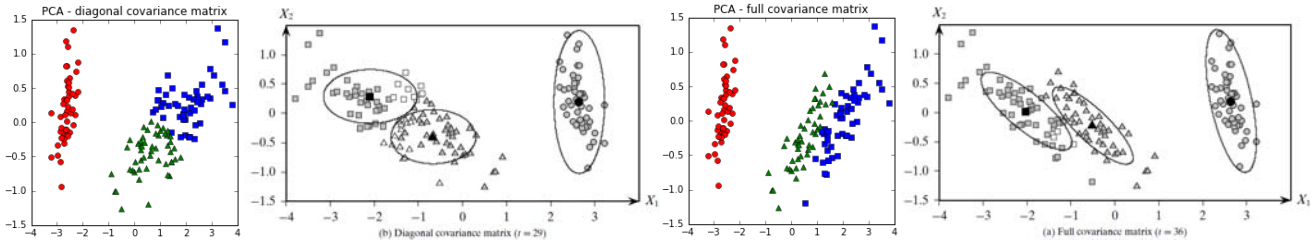
$$\Sigma_1 = \begin{pmatrix} 0.59 & 0 \\ 0 & 0.11 \end{pmatrix} \qquad \Sigma_2 = \begin{pmatrix} 0.49 & 0 \\ 0 & 0.11 \end{pmatrix} \qquad \Sigma_3 = \begin{pmatrix} 0.05 & 0 \\ 0 & 0.21 \end{pmatrix}$$

$$P(C_1) = 0.30 \qquad P(C_2) = 0.37 \qquad P(C_3) = 0.33$$

And our results:

```
mean: [[ 2.01116509  0.24112543], [ 0.6054885 -0.44341518], [-2.64084131  0.19052114]]
cov: [[[0.62 0.], [0. 0.12]], [[0.47 0.], [0. 0.10]], [[0.05 0.], [0. 0.22]]]
prior: [ 0.34598514  0.32117921  0.33333177]
```

The strange point is the number of iterations in this case (diagonal) for us is more than it from the book (37 vs. 29). I think the number of iterations in our algorithm is more rational, because in diagonal case, we have less accuracy and we need more iterations to converge to demand means.




## *Evaluating clusterings*

I have used F1 and Silhouette on PCA data with $\epsilon = 0.0001$ for $k = 2$ and $k = 3$. The results are:

| K | 2 | 3 |
|---|---|---|
| F1 | 0.83 | 0.98 |
| Silhouette | 0.71 | 0.54 |

F1 uses the real labels, so when the Euclidean distances of data points from two separate groups (clusters) are corresponding with the real labels, then the performance of F1 increases when the number of our clusters is closer to the real number of them.

In other side, Silhouette, does not use the real labels, so the only measure is Euclidean distance, and therefore the data is more compatible with the $k$s that separate data more on the Euclidean distance. Here because our PCA data is separated to two main cluster (in the Euclidean measure), we have better performance for $k = 2$.

## *Image compression*

We have selected K-mean as our compress algorithm. Besides length and width of each data point, we have also color (depth) as the third dimension – and it is why the K-mean algorithm works here at all. We can also think about the algorithms that give more weight (importance) to some particular dimensions (here color) maybe to have a better image. The reason that the image is very clear (e.g. just with $k = 4$), is that the distance measure among the colors of data points covers the distance measures among physical places of data points. So, even we have data points (pixels) that are very far from each other in the image, the algorithm connects them with the third dimension (depth). Actually, the variances among the third dimension of centers are more than variances among the first two dimensions of centers.