Problem 1

1(a)

i) Pr(P) = 0.7

ii) Pr(NP) = 0.3

iii) Pr($, yes, Korean | P) = Pr($| P) * Pr(yes | P) * Pr(Korean | P) = 3*5*2/7^3 = 30/343

iv) Pr($, yes, Korean | NP) = Pr($| NP) * Pr(yes | NP) * Pr(Korean | NP) = 1/27

1(b)

Pr(P|$, yes, korean) = pr($, yes, korean | P) * pr(P) / pr($, yes, korean)
= (30/7^3 * 0.7) / pr($, yes, korean) = (3/49)/ pr($, yes, korean)
Pr(NP | $, yes, korean) = pr($, yes, Korean | NP) * pr(NP) / pr($, yes, korean) = (1/27*0.3) / pr($, yes, korean) = (1/90) / pr($, yes, korean)
Since 3/49 > 1/ 90. The answer is P.

1(c)

For an ensemble method, we need to have several naïve bayes classifier. For each classifier, we randomly select the training samples from original training set and randomly choose a subset of all features to apply in this child classifier. Then we can train these classifiers separately depending on their own dataset and feature set. When testing, we just use majority voting method to make prediction on testing sample.

1(d)

We can use the ROC curve to evaluate. Because ROC is drawn by multiple pairs of TPR and FPR (true positive rate/ false positive rate). And the calculation is TPR = TP/P, FPR = FP/N. we can see that if the distribution of positive and negative sample is changed, then FPR and TPR would also proportionally change. Therefore, ROC curve would not be influence by the imbalance of dataset and can be used when positive examples are rare.

Problem 2

I use the code below to implement KNN. Code is in hw5.ipynb

```
: import copy
training = [[1, 0.5, 1],
            [2, 1.2, 1],
            [2.5, 2, 1],
            [3, 2, 1],
            [1.5, 2, -1],
            [2.3, 3, -1],
            [1.2, 1.9, -1],
            [0.8, 1, -1]]

testing = [[2.7, 2.7, 1],
           [2.5, 1,   1],
           [1.5, 2.5, -1],
           [1.2, 1,  -1] ]


for line in testing:
    tmp = copy.deepcopy(training)
    tmp = sorted(tmp, key = lambda sam : (sam[0]-line[0])*(sam[0]-line[0])+(sam[1]-line[1])*(sam[1]-line[1]) )
    print tmp
```

```
[[2.3, 3, -1], [2.5, 2, 1], [3, 2, 1], [1.5, 2, -1], [2, 1.2, 1], [1.2, 1.9, -1], [0.8, 1, -1], [1, 0.5, 1]]
[[2, 1.2, 1], [2.5, 2, 1], [3, 2, 1], [1.5, 2, -1], [1, 0.5, 1], [1.2, 1.9, -1], [0.8, 1, -1], [2.3, 3, -1]]
[[1.5, 2, -1], [1.2, 1.9, -1], [2.3, 3, -1], [2.5, 2, 1], [2, 1.2, 1], [3, 2, 1], [0.8, 1, -1], [1, 0.5, 1]]
[[0.8, 1, -1], [1, 0.5, 1], [2, 1.2, 1], [1.2, 1.9, -1], [1.5, 2, -1], [2.5, 2, 1], [3, 2, 1], [2.3, 3, -1]]
```

2(a)
The nearest point for(2.7, 2.7) is (2.3, 3, -1), so the prediction is -1, but the real value is 1
The nearest point for (2.5, 1) is (2, 1.2, 1), so the prediction is 1.
The nearest point for (1.5, 2.5) is (1.5, 2, -1), so the prediction is -1.
The nearest point for(1.2, 1) is (0.8, 1, -1), so the prediction is -1.
The test error is 0.25.

2(b)
The nearest point for(2.7, 2.7) is (2.3, 3,-1) and (2.5, 2, 1), so the prediction is either -1 or 1.
The nearest point for (2.5, 1) is (2, 1.2, 1) and (2.5, 2, 1), so the prediction is 1.
The nearest point for (1.5, 2.5) is (1.5, 2, -1) and (1.2, 1.9, -1), so the prediction is -1.
The nearest point for(1.2, 1) is (0.8, 1, -1) and (1, 0.5, 1), so the prediction is either -1 or 1.
So there are three situations. Both two have right prediction, or have wrong prediction, or half right half wrong.
So the test error could be 0, 0.25, 0.5.

2(c) I choose PLA.
First I initial a = 6, b = -5, c = 0.
In training set, for sample x = (1.0, 0.5), f(x) = 6*1.0-5*0.5=3.5, so the prediction is 1
for sample x = (2.0, 1.2), f(x) = 6*2.0-5*1.2=6.0, so the prediction is 1
for sample x = (2.5, 2.0), f(x) = 6*2.5-5*2.0=5.0, so the prediction is 1
for sample x = (3.0, 2.0), f(x) = 6*3.0-5*2.0=8.0, so the prediction is 1
for sample x = (1.5, 2.0), f(x) = 6*1.5-5*2.0=-1.0, so the prediction is -1
for sample x = (2.3, 3.0), f(x) = 6*2.3-5*3.0=-1.2, so the prediction is -1
for sample x = (1.2, 1.9), f(x) = 6*1.2-5*1.9=-2.3, so the prediction is -1
for sample x = (0.8, 1.0), f(x) = 6*0.8-5*1.0=-0.2, so the prediction is -1
So, the training error is 0.

In testing set, for sample x = (2.7, 2.7), f(x) = 6*2.7-5*2.7=2.7, so the prediction is 1
for sample x = (2.5, 1.0), f(x) = 6*2.5-5*1.0=10.0, so the prediction is 1
for sample x = (1.5, 2.5), f(x) = 6*1.5-5*2.5=-3.5, so the prediction is -1
for sample x = (1.2, 1.0), f(x) = 6*1.2-5*1.0=2.2, so the prediction is 1, but real value is -1.

So, the testing error is 0.25.

Finally, the a, b, c are 6, -5, 0.
the reason is that PLA can slightly change at each training point to achieve a better classifying performance.

2(d)
KNN don't have training process, it just directly use distance measure to compare each testing sample with all training set, and then get a result. Those linear methods have training process, and their training process is independent with classifying process. Though the training process would take times, but once they are trained, they usually don't need to modify again. Therefore KNN's time cost is much larger than linear methods.
KNN does not make any assumption on data, while those linear classifiers need to assume that data is linearly separable, otherwise the kernel tricks should be used.
KNN is easy to implement. And it is not complicated since it only have one parameter, k. but for those linear methods, the parameter number would vary depending on the dimension of dataset.

Problem 3

3(a)

I use the code below to implement kmeans. Code is in hw5.ipynb.

```
import copy
training = [[1, 1, 3],[2, 1, 2], [3, 2, 1], [4, 2, 2], [5, 2, 3], [6, 3, 2], [7, 5, 3], [8, 4, 3],[9, 4, 5],[10, 5, 4],
[11, 5, 5],[12, 6, 4],[13, 6, 5]]

p = [[1, 0, 3],[2, 6, 4]]
cluster = [[], []]
old_cluster = []

cnt = 0
while cnt < 3:
    for line in training:
        dist0 = (line[1]-p[0][1])*(line[1]-p[0][1])*1.0 + (line[2]-p[0][2])*(line[2]-p[0][2])*1.0
        dist1 = (line[1]-p[1][1])*(line[1]-p[1][1])*1.0 + (line[2]-p[1][2])*(line[2]-p[1][2])*1.0
        if dist0 <= dist1:
            if line not in cluster[0]: cluster[0].append(line)
            if line in cluster[1]: cluster[1].remove(line)
        else:
            if line not in cluster[1]: cluster[1].append(line)
            if line in cluster[0]: cluster[0].remove(line)

    sum_ = [0, 0]
    for point in cluster[0]:
        sum_[0] += point[1]; sum_[1] += point[2]
    p[0][1] = sum_[0]*1.0/len(cluster[0])
    p[0][2] = sum_[1]*1.0/len(cluster[0])

    sum_ = [0, 0]
    for point in cluster[1]:
        sum_[0] += point[1]; sum_[1] += point[2]
    p[1][1] = sum_[0]*1.0/len(cluster[1])
    p[1][2] = sum_[1]*1.0/len(cluster[1])
    print 'iteration: ', cnt
    print 'clus 0:', cluster[0]
    print 'center 0: ', p[0]
    print 'clus 1:', cluster[1]
    print 'center 1: ', p[1]
    cnt += 1

    if old_cluster == cluster:
        break

    old_cluster = cluster
```

The result is below.

```
iteration:  0
clus 0: [[1, 1, 3], [2, 1, 2], [3, 2, 1], [4, 2, 2], [5, 2, 3], [6, 3, 2]]
center 0:  [1, 1.8333333333333333, 2.1666666666666665]
clus 1: [[7, 5, 3], [8, 4, 3], [9, 4, 5], [10, 5, 4], [11, 5, 5], [12, 6, 4], [13, 6, 5]]
center 1:  [2, 5.0, 4.142857142857143]
iteration:  1
clus 0: [[1, 1, 3], [2, 1, 2], [3, 2, 1], [4, 2, 2], [5, 2, 3], [6, 3, 2]]
center 0:  [1, 1.8333333333333333, 2.1666666666666665]
clus 1: [[7, 5, 3], [8, 4, 3], [9, 4, 5], [10, 5, 4], [11, 5, 5], [12, 6, 4], [13, 6, 5]]
center 1:  [2, 5.0, 4.142857142857143]
```

We can see that after two iterations, the cluster centers are not changed, two clusters information are below.

Cluster 0: center: [1.83, 2.16]

Points: [1, 3], [1, 2], [2, 1], [2, 2], [2, 3], [3, 2].

Cluster 1: [5.0, 4.14].

Points: [5, 3], [4, 3], [4, 5], [5, 4], [5, 5], [6, 4], [6, 5]

This is reasonable since from Figure 2 we can intuitively see two clusters, one is up right anot her is down left. And the Kmeans clustering result is as expected.

3(b)

Firstly i choose a start point, let it be [1, 3].

Then find all density reachable points from [1, 3]. We can find [1, 2], [2, 1], [2, 2], [2, 3], [3, 2]. Since the point number > MinPts, so [1, 3] is a core object.
Then we again choose a point from the rest of unprocessed points, let it be [5, 3]
Then find all density reachable points from [5, 3]. We can find [4, 3], [4, 5], [5, 4], [5, 5], [6, 4], [6, 5].
So far, all points have been processed. Therefore we can have two clusters:
Cluster 0: [1, 3], [1, 2], [2, 1], [2, 2], [2, 3], [3, 2].
Cluster 1: [5, 3], [4, 3], [4, 5], [5, 4], [5, 5], [6, 4], [6, 5].

3(c)
Firstly we start doing clustering on single points. Since many points have 1 distance between each other, we can aggregate them.
At the beginning we have 13 clusters.
Merge [1,2] and [2,2], mark as C0 (cluster 0).
Merge [2,3] into C0.
Merge [1,3] into C0.
Merge [2,1] into C0.
Merge [3,2] into C0.
Merge [4,3] and [5,3], mark as C1 (cluster 1)
Merge [5,4] into C1.
Merge [5,5] into C1.
Merge [4,5] into C1.
Merge [6,4] into C1.
Merge [6,5] into C1.
Now we have two clusters, cluster 0 ([1, 3], [1, 2], [2, 1], [2, 2], [2, 3], [3, 2]), and cluster 1 ([5, 3], [4, 3], [4, 5], [5, 4], [5, 5], [6, 4], [6, 5]).
Finally we can merge C0 and C1.

I use the code below to perform AGNES. Code is in hw5.ipynb.

```
from scipy.cluster.hierarchy import dendrogram, linkage
from matplotlib import pyplot as plt
X = [[1, 3], [1, 2], [2, 1], [2, 2], [2, 3], [3, 2], [5, 3], [4, 3], [4, 5], [5, 4], [5, 5], [6, 4], [6, 5]]

Z = linkage(X, 'single')
fig = plt.figure(figsize=(25, 10))
dn = dendrogram(Z)
plt.show()
```

The result is below.