Linear Classification and the Binomial Distribution

Gruppe:
MeSi
Autoren:
Jens Meiners
Arne Siebenmorgen

7.1 Linear Discriminant Analysis (3 points)

A popular linear classifier known as *linear discriminant analysis* (LDA) can be motivated from conditional Gaussian density estimation for classes with equal covariances: Assume that samples from class c are drawn according to the Gaussian density function

$$p(\vec{x}|c) = \frac{1}{(2\pi)^{d/2}|\mathbf{\Sigma}|^{1/2}} \exp\left(-\frac{1}{2}(\vec{x}-\vec{\mu}_c)^\top \mathbf{\Sigma}^{-1}(\vec{x}-\vec{\mu}_c)\right) ,$$

where $\vec{\mu}_c$ is the *conditional mean* of class c, and $\mathbf{\Sigma}$ is the common covariance matrix. This allows to determine the conditional probability of the classes given the observed data and the classifier selects the class with the highest conditional probability.

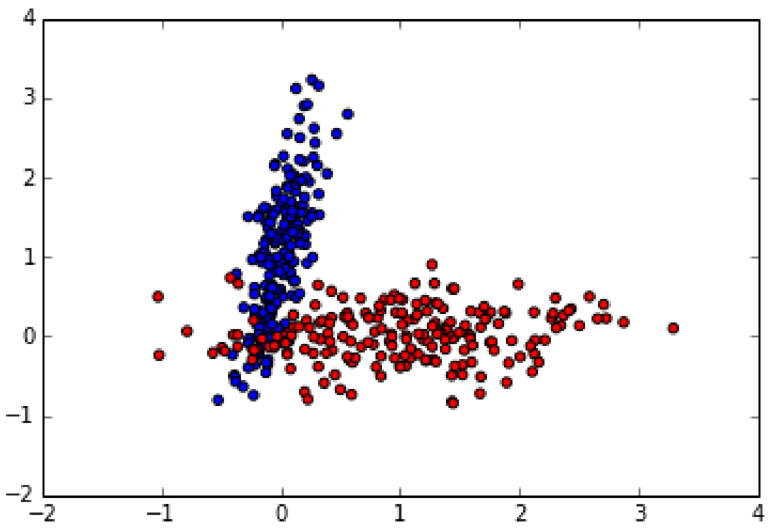
For the case of 2 classes, the decision boundary of the LDA classifier can be expressed as $\vec{w}^\top\vec{x} - b = 0$. Determine the parameters $\vec{w} \in \mathbb{R}^d$ and $b \in \mathbb{R}$. Can you say what the shape of the boundary looks like for the case of unequal covariances?

In [1]:

```
# prepare data
import numpy as np
import matplotlib.pyplot as plt
import matplotlib as mpl
from matplotlib import colors
%matplotlib inline
```

In [2]:

```
cov1 = [[0.1, 0.7],
       [0, 0.1]]
cov2 = [[0.7, 0],
       [0.1, 0.1]]

mean1 = [0. , 1.]
mean2 = [1. , 0.]

X1,y1 = np.random.multivariate_normal(mean1, cov1, 200).T
X2,y2 = np.random.multivariate_normal(mean2, cov2, 200).T

plt.scatter(X1, y1, c='blue')
plt.scatter(X2, y2, c='red')
plt.show()
```



In [3]:

```
from sklearn.lda import LDA
```

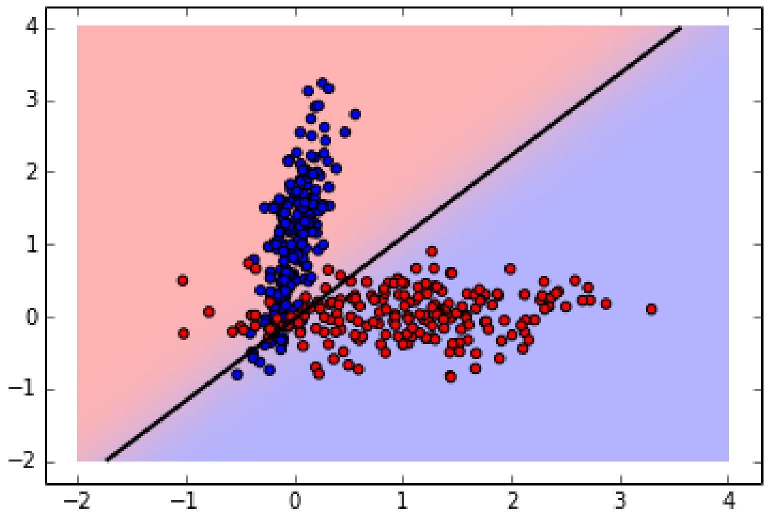
In [4]:

```
X = np.vstack((np.hstack((X1,X2)),np.hstack((y1,y2)))).T
y = np.ones(X.shape[0])
y[:X.shape[0]/2] *= -1

lda = LDA(n_components=2)
X_lda = lda.fit_transform(X,y)

print 'weights:\n',lda.coef_
print 'b:\n',lda.intercept_
```

```
weights:
[[-1.29831382]
 [ 1.29831382]]
b:
[-1.53595657 -1.53595657]
```

In [5]:

```
cmap = colors.LinearSegmentedColormap(

    'red_blue_classes',
    {'blue': [(0, 0.7, 0.7), (1, 1, 1)],
     'green': [(0, 0.7, 0.7), (1, 0.7, 0.7)],
     'red': [(0, 1, 1), (1, 0.7, 0.7)]})
def plot_fisher(model):
    plt.cm.register_cmap(cmap=cmap)

    plt.scatter(X1, y1, c='blue')
    plt.scatter(X2, y2, c='red')

    # class 0 and 1 : areas
    nx, ny = 200, 100
    x_min, x_max = plt.xlim()
    y_min, y_max = plt.ylim()
    xx, yy = np.meshgrid(np.linspace(x_min, x_max, nx),
                         np.linspace(y_min, y_max, ny))
    Z = model.predict_proba(np.c_[xx.ravel(), yy.ravel()])
    Z = Z[:, 1].reshape(xx.shape)
    plt.pcolormesh(xx, yy, Z, cmap='red_blue_classes',
                   norm=colors.Normalize(0., 1.))
    plt.contour(xx, yy, Z, [0.5], linewidths=2., colors='k')

    plt.scatter(X1, y1, c='blue')
    plt.scatter(X2, y2, c='red')

    plt.show()
plot_fisher(lda)
```



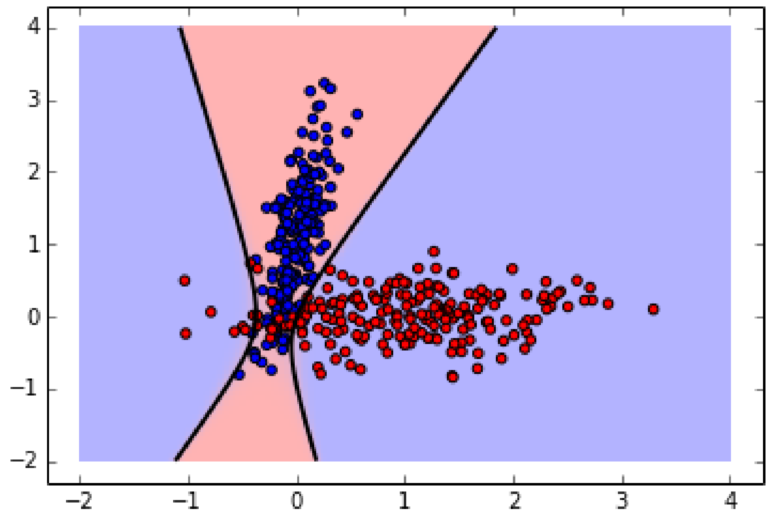
As seen in the previous plot, LDA on unequal covariance datasets.
LDA can only learn linear boundries. quadratic lda is more flexible, see the next example

In [6]:

```
from sklearn.qda import QDA
```

In [7]:

```
qda = QDA()
qda.fit(X,y)

plot_fisher(qda)
```



We have seen that LDA yields good results for equal covariances. As the example above showed, this may not hold true if the covariances differ. QDA was proposed as a model to induce more flexibility to LDA.

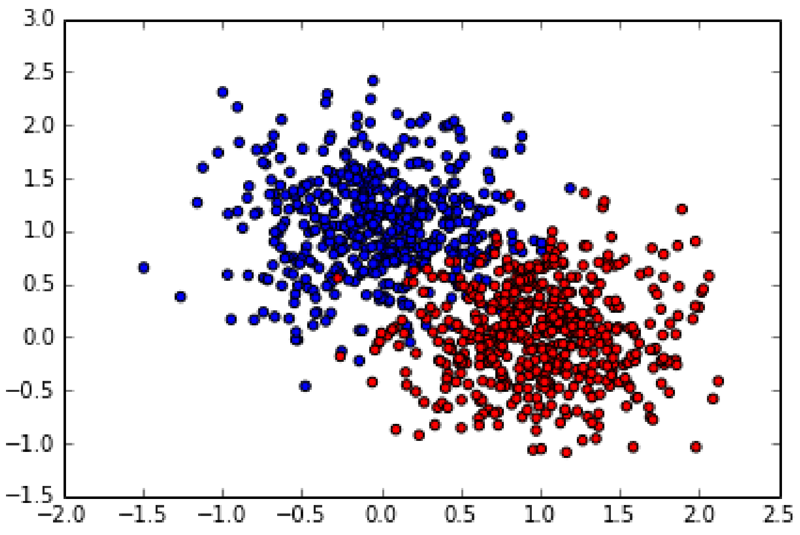
7.2 Variability of classification (4 points)

Assume data $x_\alpha \in \mathbb{R}^2$ drawn from two clusters, C_1 and C_2 and distributed according to the (multivariate) Normal distributions $\mathcal{N}(\mu_i, \sigma)$, $i = 1, 2$ with $\mu_1 = (0, 1)$, $\mu_2 = (1, 0)$, and $\sigma = 2I_2$ where I_2 is the 2x2 identity matrix. This task examines, how well a linear connectionist neuron can separate these two classes for increasing amounts N of available training data. Proceed as follows:

1. Generate a sample of $N/2$ data $\vec{x}_\alpha$ from each of the two clusters. Let $y_\alpha = \pm 1$ for $\vec{x}_\alpha$ from C_1 and C_2, respectively.

In [91]:

```
def gen_data(N, plot=False):
    cov = np.eye(2)*0.2

    mean1 = [0. , 1.]
    mean2 = [1. , 0.]

    X1,y1 = np.random.multivariate_normal(mean1, cov, N/2).T
    X2,y2 = np.random.multivariate_normal(mean2, cov, N/2).T

    if plot:
        plt.scatter(X1, y1, c='blue')
        plt.scatter(X2, y2, c='red')
        plt.show()


    X = np.hstack((np.vstack((X1,y1)),np.vstack((X2,y2)))).T
    y = np.ones(X.shape[0])
    y[:X.shape[0]/2] *= -1
    return X,y
dummy = gen_data(1000,True)
```



2. Find the weights of a linear connectionist neuron with output $y_\alpha = \vec{w}^\top \vec{x}_\alpha + b$ minimizing the squared error according to the analytical formula (see Problem 4.2a in Ex. 4).

In [92]:

```
from numpy.linalg import lstsq
```

In [251]:

```
X,y = gen_data(1000)
A = np.vstack([X[:,0],np.ones(len(X[:,0]))]).T
w = lstsq(A,y)[0]
```

3. Find the predictions $\hat{y} = \text{sign}(\vec{w}^\top \vec{x} + b)$ of this classifier for $N_{\text{test}} = 1000$ new data drawn from the same distributions.

In [253]:

```
X_test, y_test = gen_data(1000)
pred = np.sign(w.dot(X_test.T)) == y_test
```

4. Calculate the percentage of correct classifications for the training (r_{train}) and test samples (r_{test}).

In [254]:

```
print pred.sum()/float(len(pred))
```

```
0.911
```

For $N = 2, 4, 6, 8, 10, 20, 40, 100$:

(a) Repeat these steps 50 times and save the resulting parameters as well as the percentages for training and testing.

In [246]:

```
Ns = [2,4,6,8,10,20,40,100]
params = np.zeros((len(Ns),50,2))
perc_train = np.zeros((len(Ns), 50))
perc_test = np.zeros((len(Ns), 50))
for i, N in enumerate(Ns):
    for j in range(50):
        X, y = gen_data(N)
        A = np.vstack([X[:,0],np.ones(len(X[:,0]))]).T
        w = lstsq(A,y)[0]
        params[i,j] = w
        X_test, y_test = gen_data(1000)
        pred = np.sign(w.dot(X.T)) == y
        perc_train[i,j] = pred.sum()/float(len(pred))
        pred = np.sign(w.dot(X_test.T)) == y_test
        perc_test[i,j] = pred.sum()/float(len(pred))
```

(b) For $w_1, w_2, r_{\text{train}}$, and r_{test}, plot the mean values and standard deviations (plotting N on the x-axis and the corresponding statistic on the y-axis).

In [247]:

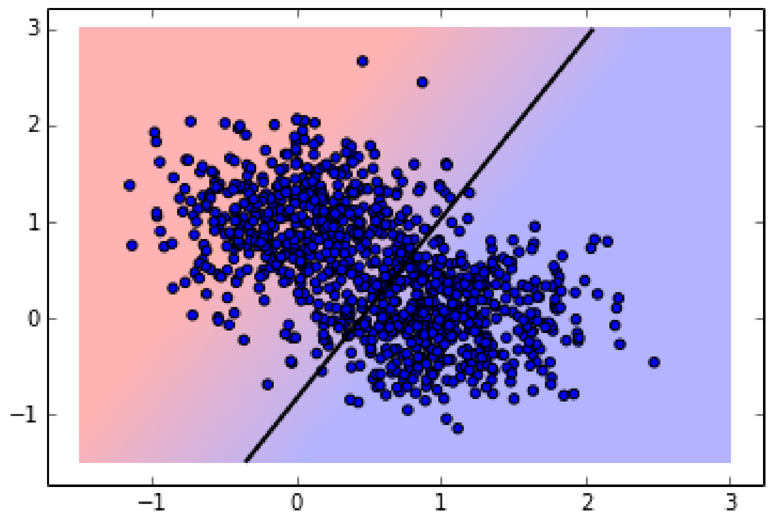
```
print 'overall means, to see if everything makes sense'

means_per_N = params.mean(axis=1)
means_overall = means_per_N.mean(axis=0)
print 'w_1:',means_overall[0],'\tw_2:',means_overall[1]
print 'r_train:',perc_train.mean(),'\tr_test:',perc_test.mean()
```

```
overall means, to see if everything makes sense
w_1: 2.52103637734      w_2: -1.71146222631
r_train: 0.914345833333         r_test: 0.8917125
```

In [248]:

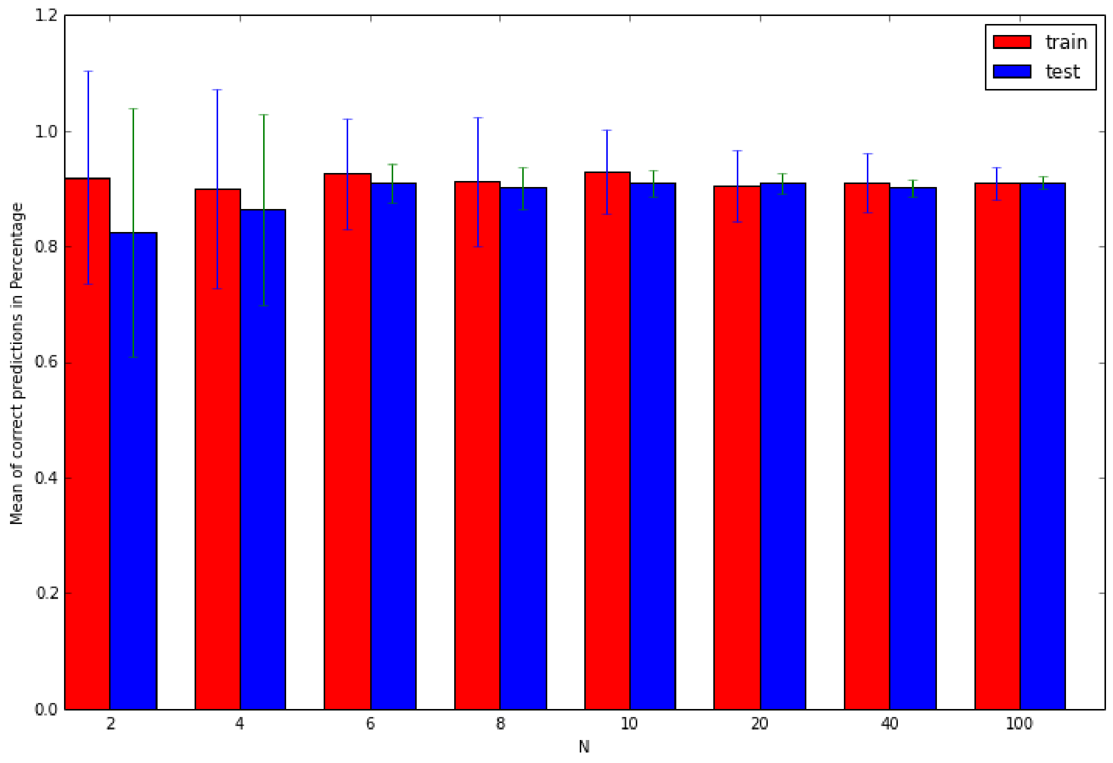
```
##############################
# first of all, plot the decision line to make sense of our numbers

X,y = gen_data(1000)
plt.scatter(X[:,0], X[:,1])
A = np.vstack([X[:,0], np.ones(len(X[:,0]))]).T
w = lstsq(A,y)[0]
print w
pred = np.sign(w.dot(X.T)) == y
print pred.sum()/float(len(pred))

xi = np.arange(-1,3)
plt.cm.register_cmap(cmap=cmap)

nx, ny = 200, 100
x_min, x_max = plt.xlim()
y_min, y_max = plt.ylim()
xx, yy = np.meshgrid(np.linspace(x_min, x_max, nx),
                     np.linspace(y_min, y_max, ny))

Z = w.dot(np.array([xx.ravel(), yy.ravel()]))
Z = Z.reshape(xx.shape)
plt.pcolormesh(xx, yy, Z, cmap='red_blue_classes',
               norm=colors.Normalize(-1,1))
plt.contour(xx, yy, Z, [0.5], linewidths=2., colors='k')
plt.scatter(X[:,0], X[:,1])
plt.show()
```

```
[ 1.11005987 -0.59373732]
0.903
```



In [264]:

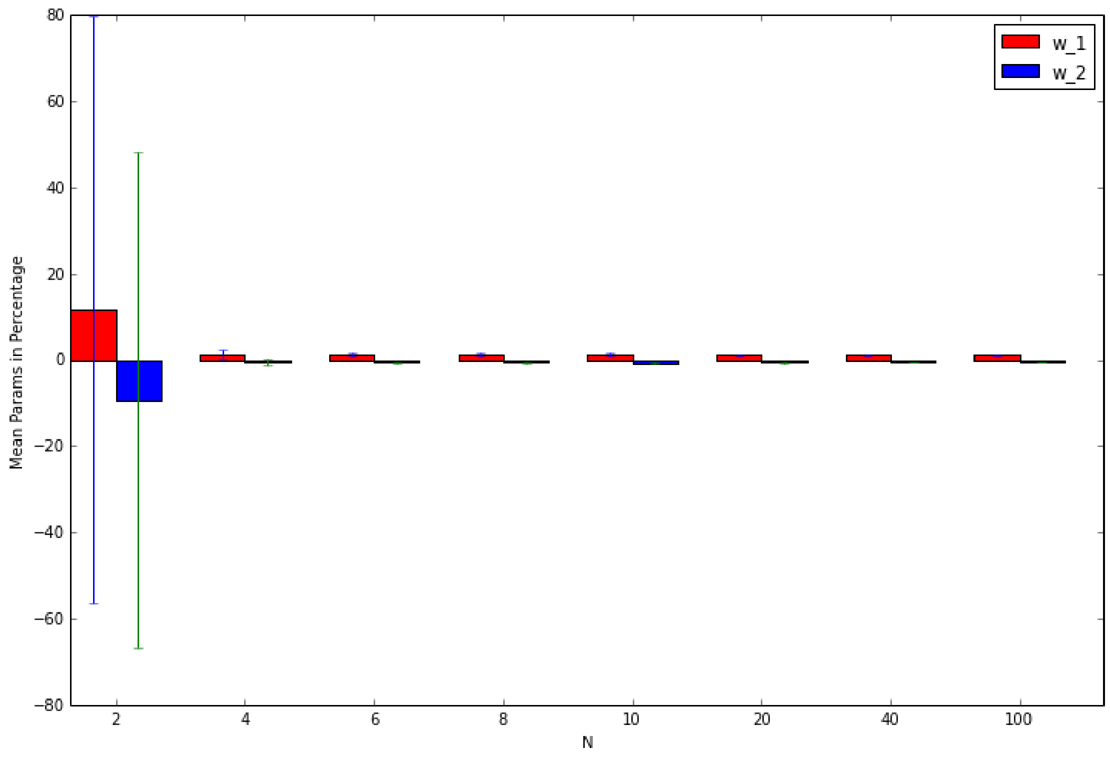
```
fig, ax = plt.subplots()
fig.set_figheight(8)
fig.set_figwidth(12)
ix = np.arange(len(Ns))
width = 0.35
bar1 = ax.bar(ix, perc_train.mean(axis=1), width, color='r', yerr=perc_tra
in.std(axis=1))
bar2 = ax.bar(ix + width, perc_test.mean(axis=1), width, color='b', yerr=p

erc_test.std(axis=1))
ax.set_ylabel('Mean of correct predictions in Percentage')
ax.set_xlabel('N')
ax.set_xticks(ix + width)
ax.set_xticklabels([str(n) for n in Ns])
ax.legend((bar1[0], bar2[0]), ('train', 'test'))
plt.show()
print ['N='+str(Ns[i])+' test_error: '+str(e) for i,e in enumerate(perc_te
st.mean(axis=1))]
```



```
['N=2 test_error: 0.82492', 'N=4 test_error: 0.86422', 'N=6 t
est_error: 0.90942', 'N=8 test_error: 0.90226', 'N=10 test_er
ror: 0.90946', 'N=20 test_error: 0.90984', 'N=40 test_error:
0.90254', 'N=100 test_error: 0.91104']
```

In [263]:

```
means_per_N = params.mean(axis=1)
stds_per_N = params.std(axis=1)
fig, ax = plt.subplots()
fig.set_figheight(8)
fig.set_figwidth(12)
ix = np.arange(len(Ns))
width = 0.35
bar1 = ax.bar(ix, means_per_N[:,0], width, color='r', yerr=stds_pe
r_N[:,0])

bar2 = ax.bar(ix + width, means_per_N[:,1], width, color='b', yerr=stds_pe
r_N[:,1])
ax.set_ylabel('Mean Params in Percentage')
ax.set_xlabel('N')
ax.set_xticks(ix + width)
ax.set_xticklabels([str(n) for n in Ns])
ax.legend((bar1[0], bar2[0]), ('w_1', 'w_2'))
plt.show()
print ['N='+str(Ns[i])+' test_error: '+str(e) for i,e in enumerate(stds_pe
r_N)]
```



```
['N=2 test_error: [ 68.21595985  57.50726512]', 'N=4 test_err
or: [ 1.05359264  0.68230829]', 'N=6 test_error: [ 0.32608658
0.29637733]', 'N=8 test_error: [ 0.32945378  0.28022417]',
'N=10 test_error: [ 0.30203243  0.30837152]', 'N=20 test_erro
r: [ 0.2124418   0.16049171]', 'N=40 test_error: [ 0.09863117
0.07746856]', 'N=100 test_error: [ 0.07025814  0.05475211]']
```

(c) Interpret your results! How do these estimates depend on N?

test error minimizes for $max(N)$. Standard deviations also minimize for $max(N)$.
If trained on a small dataset, the model naturally failes to generalize well. This is due to its high standart deviation. Since there are not enough points, the gradient and offset of the decision boundry are not very stable. As the datasize grows, the parameters are more and more stabalized: std. dev. and percentage error decrease.
Because the data is not linearly seperable, a percentage error of around $10\,\%$, for high N, as we encountered seems to be quite good.

This behavior, to us, seems to be not of a surprise, since the $test$ set is distributed in exactly the same way as the $train$ set. Therefor, we have a good impression on the generalization of this specific, generated, problem. For a closer look on real data, having a much more complex, maybe shifting distribution, this results are not as helpful as they might seem to be.

7.3 Approximations to the Binomial distribution (3 points)

This exercise examines the relation between the following 3 distributions:

$$f(k; n, p) = \binom{n}{k} p^k (1-p)^{n-k} \qquad \text{(Binomial distribution)}$$

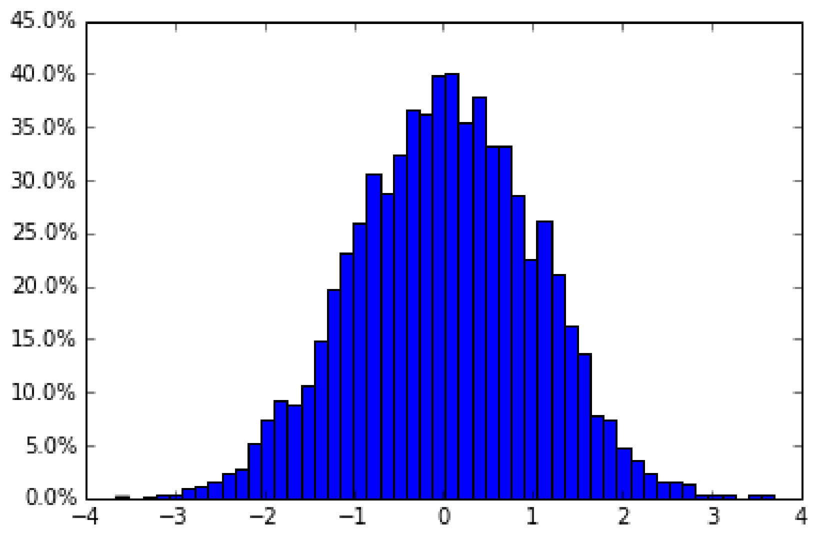
$$f(x; \mu, \sigma) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}} \qquad \text{(Normal distribution)}$$

$$f(k; \lambda) = \frac{\lambda^k}{k!} e^{-\lambda} \qquad \text{(Poisson distribution)}$$

(a) Visualize the probability mass function $f(k; n, p)$ of the binomial distribution for a few different values of k, n, p. Describe an example random experiment, for which the binomial distribution might be a good model. What are the central properties of the binomial distribution and in which situations might it therefore not be a good model?

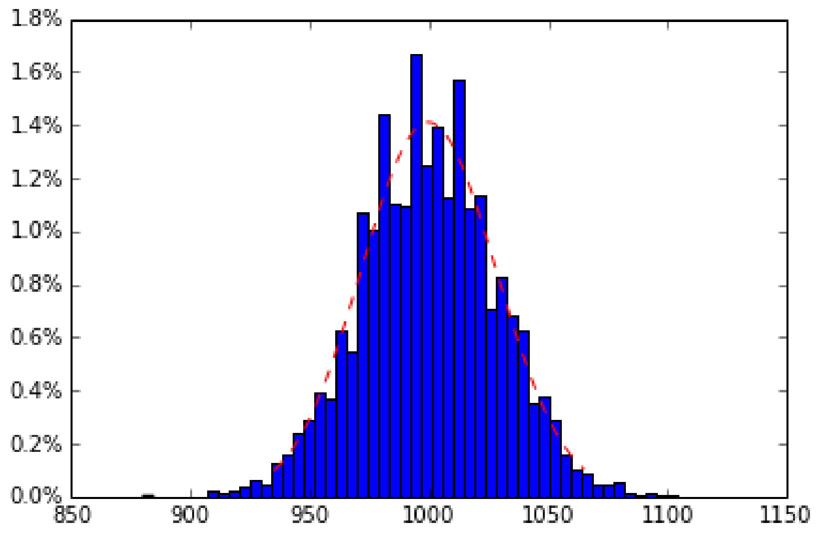
In [329]:

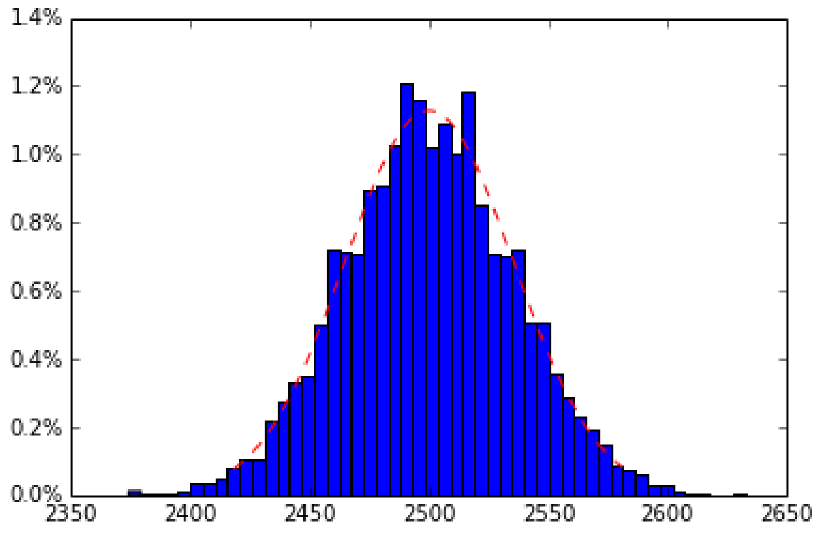
```
##################################
# first of all, implement a method to plot the distributions

import matplotlib
from matplotlib.ticker import FuncFormatter

def to_percent(y, position):
    # Ignore the passed in position. This has the effect of scaling the de
fault
    # tick locations.
    s = str(100 * y)

    # The percent symbol needs escaping in latex
    if matplotlib.rcParams['text.usetex'] is True:
        return s + r'$\%$'
    else:
        return s + '%'

def plot_dist(x=np.random.randn(5000), bins=50, normed=True):
    # Make a normed histogram. It'll be multiplied by 100 later.
    plt.hist(x, bins, normed=normed)

    # Create the formatter using the function to_percent. This multiplies
all the
    # default labels by 100, making them all percentages
    formatter = FuncFormatter(to_percent)

    # Set the formatter
    plt.gca().yaxis.set_major_formatter(formatter)

plot_dist()
plt.show()
```

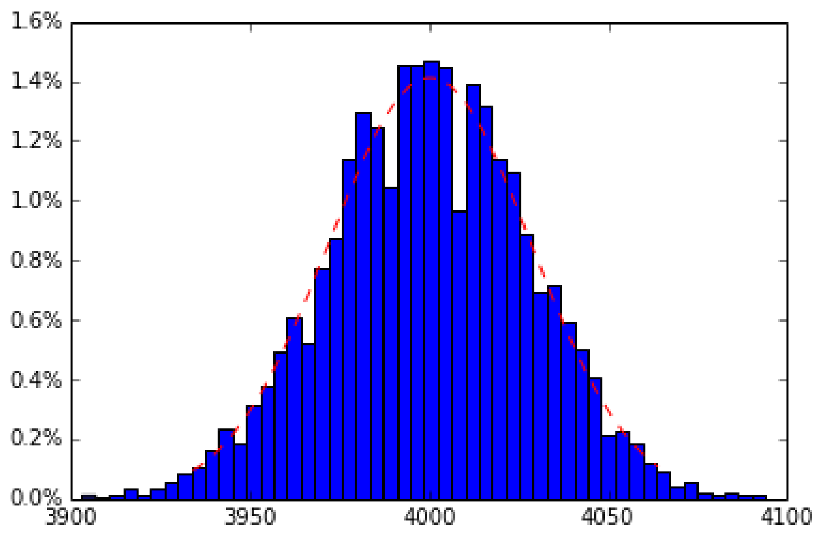


In [358]:

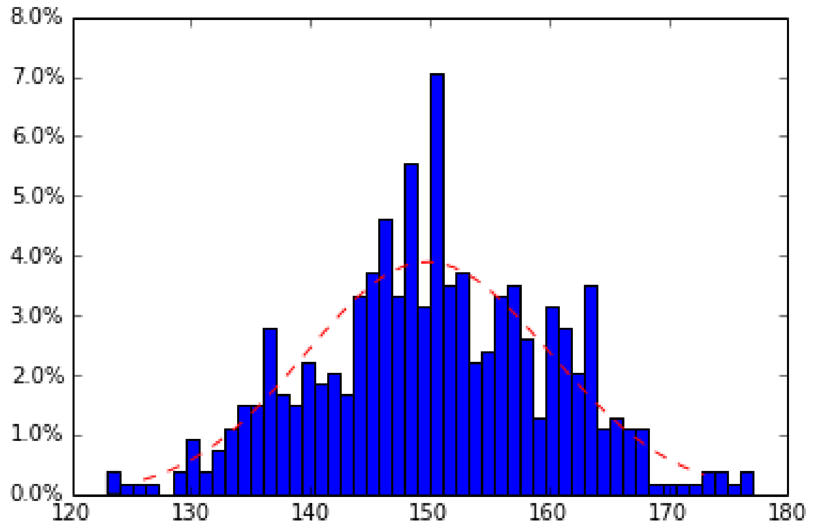
```
##################
# now for the binomial distribution

def test_binom(n=5000,p=0.3):
    x = np.arange(binom.ppf(0.01, n, p), binom.ppf(0.99, n, p))

    plot_dist(np.random.binomial(n,p,n))
    plt.plot(x,binom.pmf(x,n,p),'r--')
    plt.show()

test_binom(5000,0.2)
test_binom(5000,0.5)
test_binom(5000,0.8)
print '---------------------------------------------'
test_binom(500,0.3)
test_binom(5000,0.3)
test_binom(500000,0.3)
```



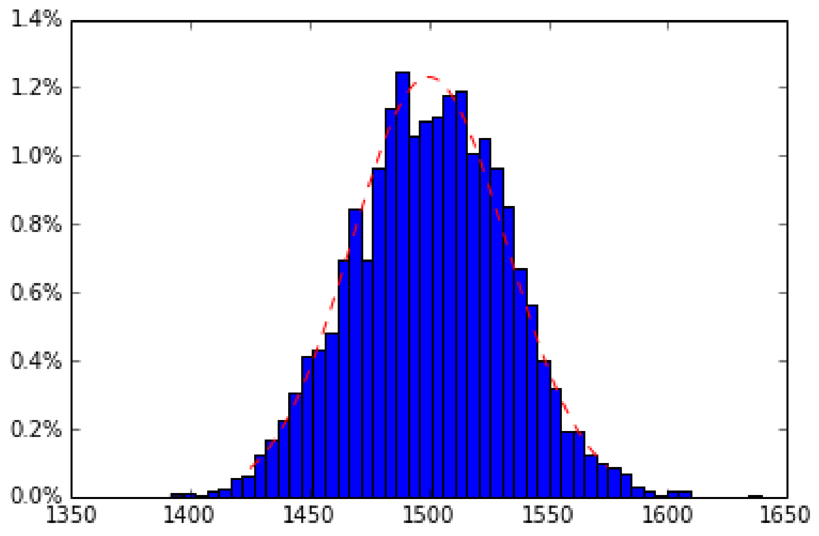




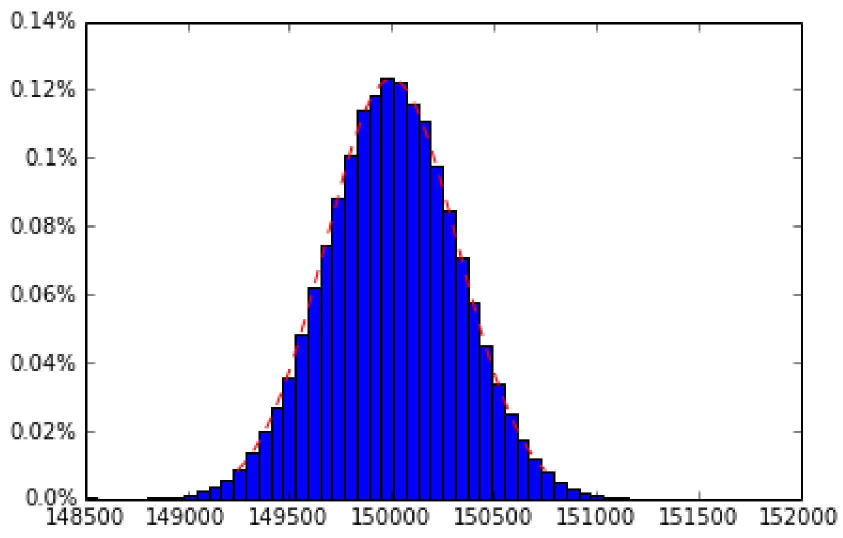
1.6%
1.4%
1.2%
1.0%
0.8%
0.6%
0.4%
0.2%
0.0%
3900
3950
4000
4050
4100



8.0%
7.0%
6.0%
5.0%
4.0%
3.0%
2.0%
1.0%
0.0%
120
130
140
150
160
170
180



1.4%
1.2%
1.0%
0.8%
0.6%
0.4%
0.2%
0.0%
1350
1400
1450
1500
1550
1600
1650



(b) The normal distribution is sometimes used as an approximation to the binomial distribution. Under which conditions is this reasonable. Under which conditions is it problematic? Visualize one example where the Normal approximation is good and one where it is not. Give at least one reason why this distribution is so widely used. Is it a good approximation for the example random experiment you gave above?

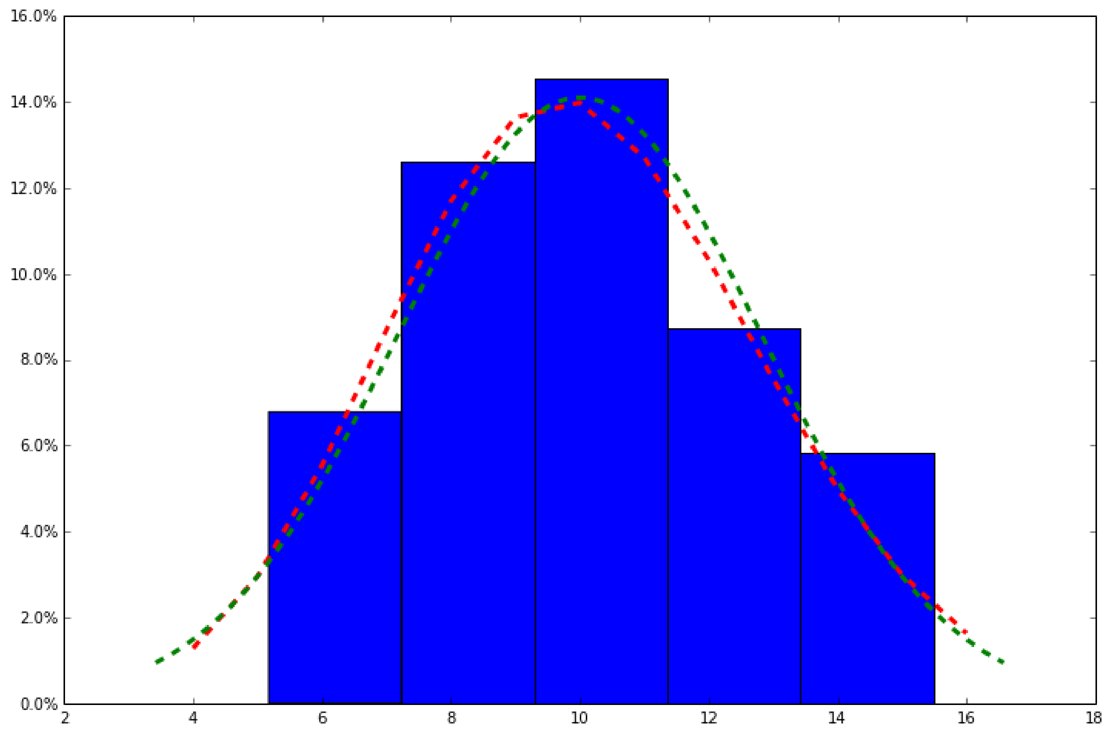
In [364]:

```
from scipy.stats import norm
```

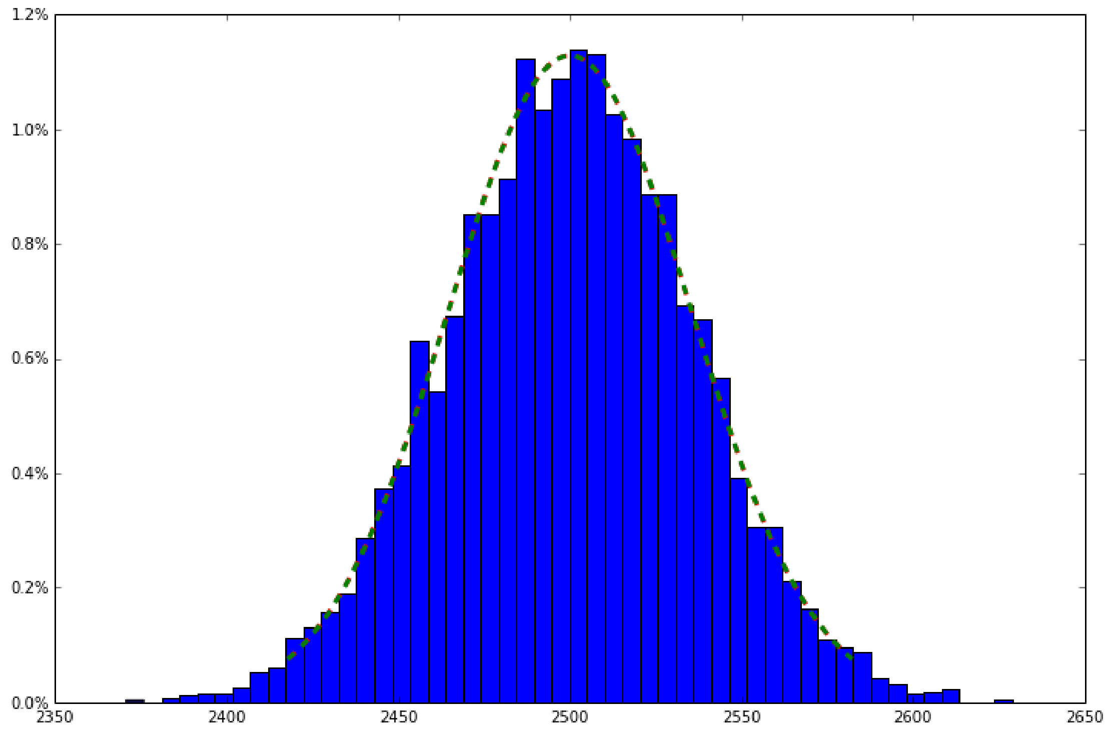
In [376]:

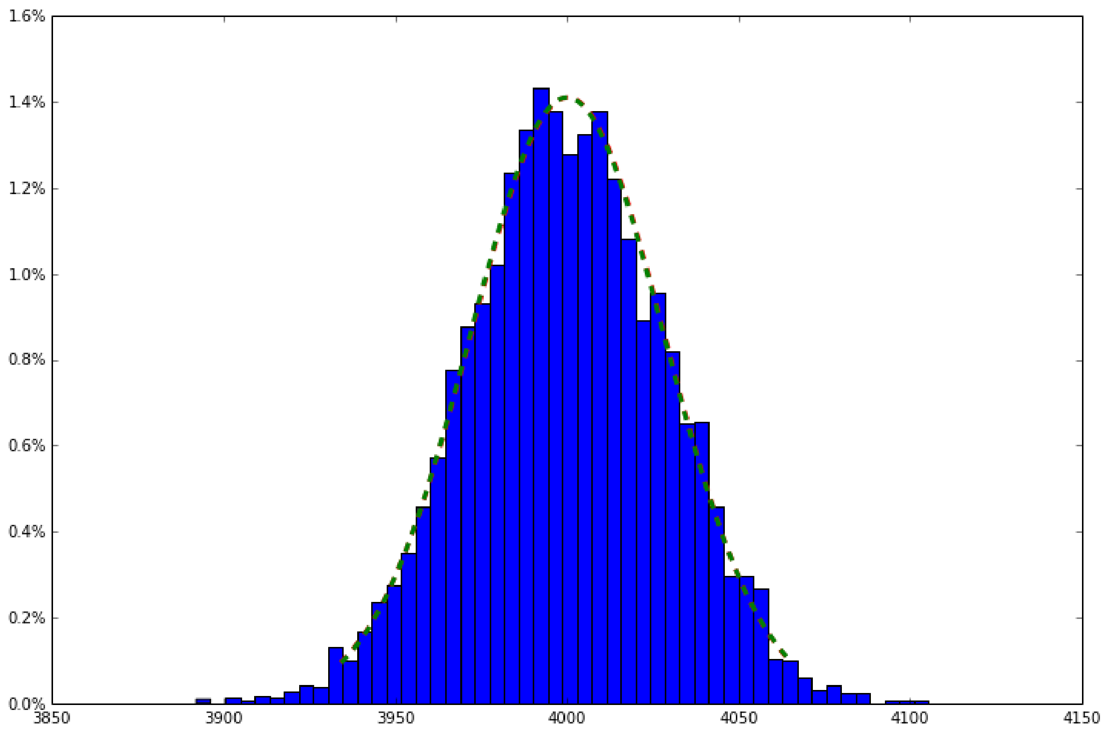
```
###################
# now for the normal distribution vs binomial

def test_normal(mu=2500,s=0.3,n=5000):
    plot_dist(np.random.normal(mu,s,n))
    plt.show()

def test_normal_vs_binom(n=5000,p=0.3, bins=50):
    plt.figure(figsize=(12, 8))
    # plot binomial dist. as dotted red line
    x = np.arange(binom.ppf(0.01, n, p), binom.ppf(0.99, n, p))
    plt.plot(x,binom.pmf(x,n,p),'r--',lw=3)

    # translate into normal variables
    mu = n*p
    sigma = np.sqrt(n*p*(1-p))
    # plot normal dist. as histogram
    plot_dist(np.random.normal(mu,sigma,n), bins=bins)
    # plot histogram of normal dist
    x = np.linspace(norm.ppf(0.01,loc=mu, scale=sigma), norm.ppf(0.99,lo
c=mu, scale=sigma), 100)
    plt.plot(x, norm.pdf(x,loc=mu, scale=sigma),'g--', lw=3, label='norm p
df')
    plt.show()

print 'red line is the binomial distribution. for small n, there are some
problems'
test_normal_vs_binom(50,0.2,bins=5)

print 'for larger n, the dottet lines are almost similar'
test_normal_vs_binom(5000,0.5)
test_normal_vs_binom(5000,0.8)
```

```
red line is the binomial distribution. for small n, there are
some problems
```



```
for larger n, the dottet lines are almost similar
```





The plots show cases where the densityfunctions are are very similar despite different probabilities.

The translation of binomial to normal parameters is as follows
normal : binomial

$$\mu = np$$
$$\sigma = \sqrt{np(1-p)}$$

One reason to do this approximation is, that the binomial distribution is discrete, while the normal distribution is continuous. This property can come in very handy if it is needed to approximate a small amount of binomial samples. Another reason is, that the skew of the binomial distribution is not too greate for large n

(c) The Poisson distribution is often used as an alternative approximation to the binomial distribution. Under which conditions is it a good approximation. Visualize one example parametrization where the Poisson approximation is good and one where it is not. Is it a good approximation for the random experiment above?

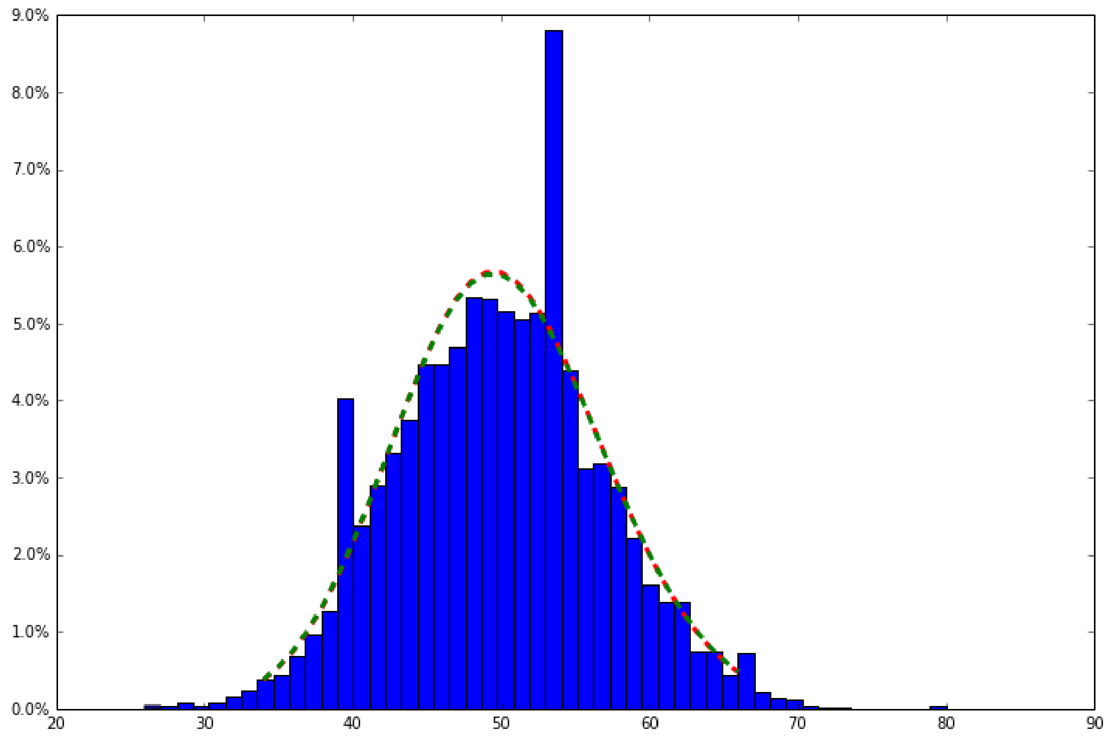
In [377]:

```
from scipy.stats import poisson
```

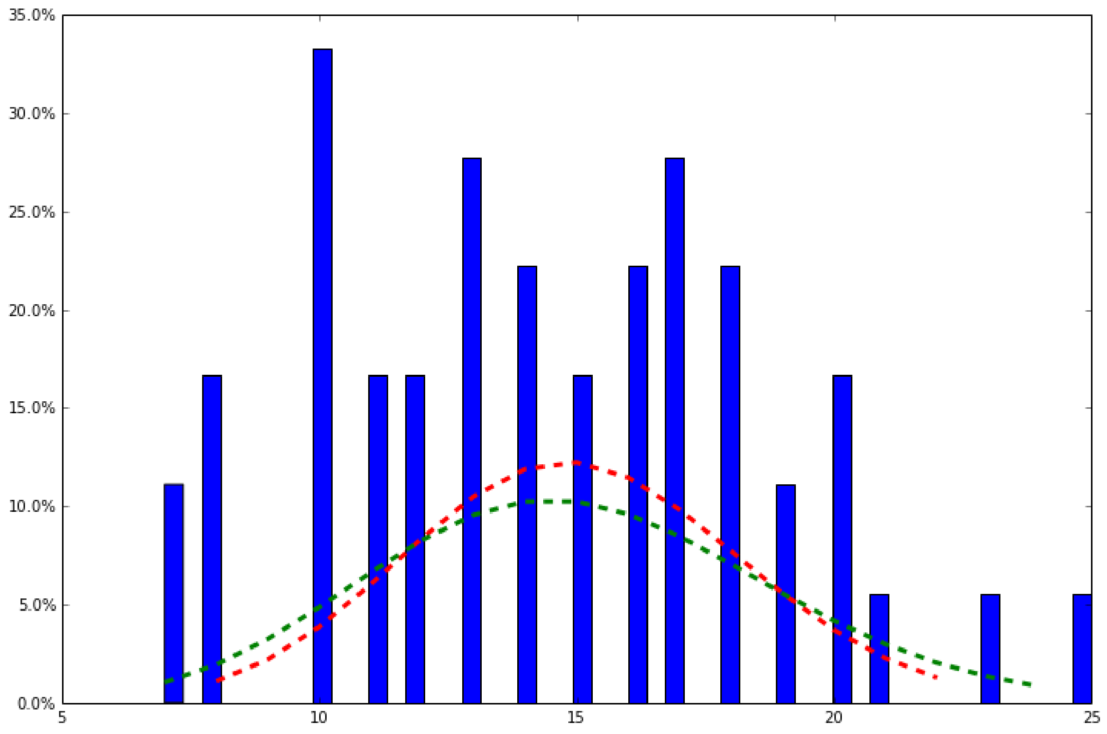
In [394]:

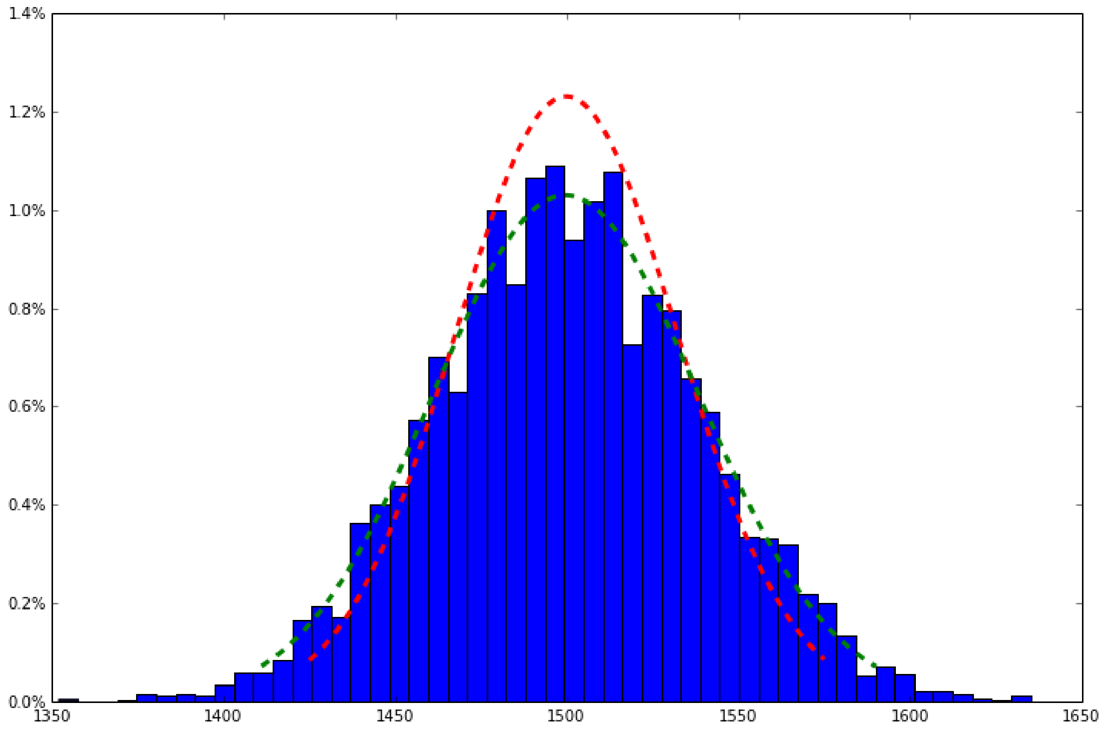
```
###################
# now for the poisson distribution vs binomial

def test_poisson(lamda=1,n=5000):
    plot_dist(np.random.poisson(lamda,n))
    plt.show()

def test_poisson_vs_binom(n=5000,p=0.3, bins=50):
    plt.figure(figsize=(12, 8))

    # plot binomial dist. as dotted red line
    x = np.arange(binom.ppf(0.01, n, p), binom.ppf(0.99, n, p))
    plt.plot(x,binom.pmf(x,n,p),'r--',lw=3)

    # translate into poisson variables
    lamb = n*p
    mu = 0
    # plot poisson dist. as histogram
    plot_dist(np.random.poisson(lamb,n), bins=bins)
    # plot poisson dist
    x = np.arange(poisson.ppf(0.01, lamb),poisson.ppf(0.99, lamb))
    plt.plot(x, poisson.pmf(x,lamb, loc=mu),'g--', lw=3, label='poisson pd
f')
    plt.show()

print 'for a big n, and small p, the two lines become more and more equal'
test_poisson_vs_binom(5000,0.01,bins=50)
print 'in other cases however, the results seem to be rediculous'
test_poisson_vs_binom(50,0.3)
test_poisson_vs_binom(5000,0.3)
```

```
for a big n, and small p, the two lines become more and more
equal
```



```
in other cases however, the results seem to be rediculous
```





First, the translation:
poisson : binomial

$$\lambda = np$$

As seen in the plots, the poisson distribution approximates the binomial if the number of trailes gets as large as possible.
NIST/SEMATECH, "6.3.3.1. Counts Control Charts", e-Handbook of Statistical Methods. stated, that: if $n \geq 20$ and $p \leq 0.05$, or if $n \geq 100$ and $np \leq 10$ this results in a good approximation