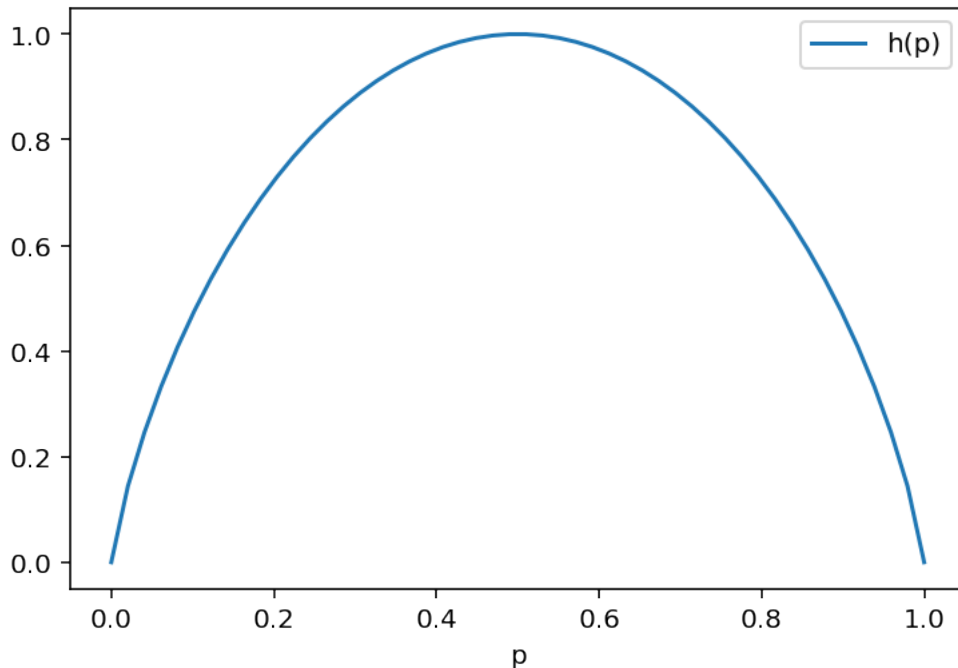
01-practice

September 12, 2019

[1]:
```
%matplotlib inline
import numpy as np
import matplotlib.pyplot as plt
from scipy.special import binom, entr
```

In this notebook we'll try to get some intuition for lossy compression and Shannon's theorem for a sequence of Bernoulli(p) random variables. The entropy of such a random variable as a function of p is given as:

[2]:
```
def h(p):
    """Return binary Shannon entropy of distribution {p,1-p}."""
    return (entr(p) + entr(1 - p)) / np.log(2)


ps = np.linspace(0, 1)
plt.plot(ps, h(ps), label='h(p)')
plt.xlabel('p')
plt.legend()
plt.show()
```



So, for $p = 0$ or $p = 1$ the entropy is 0, and there is no information (in the sense that there is only one possible outcome, so we don't have to send any bits to communicate the information). For $p = \frac{1}{2}$ the entropy is maximal and given by 1, which means by Shannon's theorem that there is no way to compress this source (it has no redundancy).

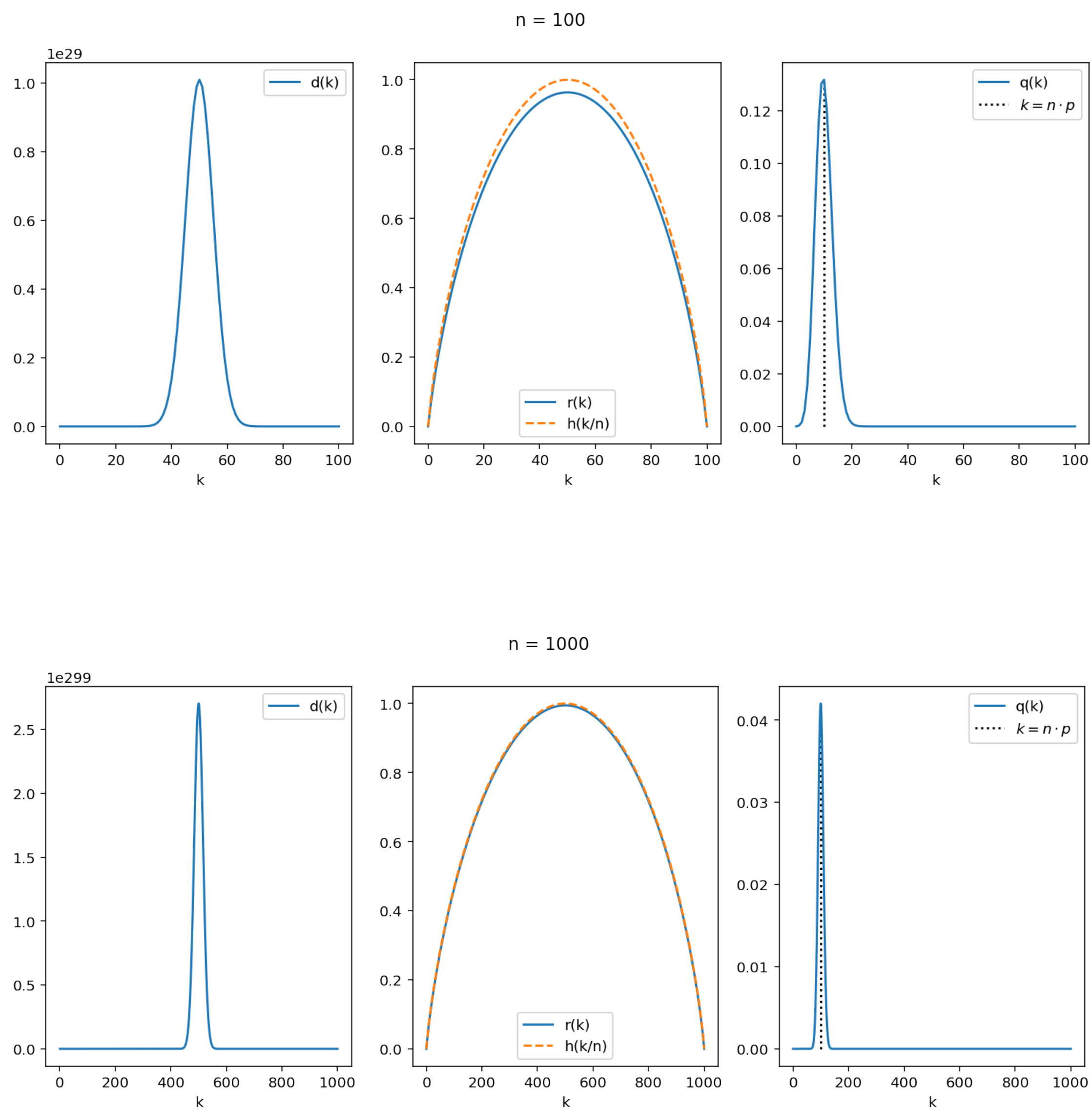
We will consider a sequence of Bernoulli random variables, and we will see what happens if we try to compress using typical sets. First of all, notice that the probability of a sequence only depends on the number k of ones in the sequence. We want to see that the probability of k peaks strongly around $k \approx p \cdot n$. Let

$$d(k) = \binom{n}{k}, \quad r(k) = \frac{1}{n} \log \binom{n}{k}, \quad q(k) = \binom{n}{k} p^k (1-p)^{n-k}$$

so $q(k)$ is the probability of having k ones. We plot these functions for $k \in \{0, 1, \ldots, n\}$ and $n = 100, 1000$ for $p = 0.1$.

[3]:
```
p = 0.1

for n in [100, 1000]:
    ks = np.arange(n + 1)
    ds = binom(n, ks)
    rs = np.log2(ds) / n
    qs = ds * p**ks * (1 - p)**(n - ks)

    fig, axes = plt.subplots(ncols=3, figsize=(12.8, 4.8))
    fig.suptitle('n = %s' % n)
    axes[0].plot(ks, ds, label='d(k)')
    axes[0].set_xlabel('k')
```

```
    axes[0].legend()

    axes[1].plot(ks, rs, label='r(k)')
    axes[1].plot(ks, h(ks / n), '--', label='h(k/n)')
    axes[1].set_xlabel('k')
    axes[1].legend()

    axes[2].plot(ks, qs, label='q(k)')
    axes[2].vlines(
        p * n, 0, np.max(qs), linestyle='dotted', label='$k = n \cdot p$')
    axes[2].set_xlabel('k')
    axes[2].legend()
```



We see two important features: - The sets of sequences of length n with fixed number k of ones have cardinality $2^{n(h(k/n)\pm o(1))}$ (since $r(k)$ is approximately $h(k/n)$). - The probability that a random sequence has k ones is peaked about $k/n \approx p$.

The typical set will consist of sequences which have k ones such that k/n is close to p. From the above pictures we have good hope that the probability of being in such a set is large (for large n), while the size of such sets is significantly smaller than the set of all sequences ($2^{n(h(k/n)\pm o(1))}$ versus 2^n). To verify this we will plot the logarithm of the size of the typical set and the probability of being in the typical set

$$r(n) = \frac{1}{n}\log|T_{n,\varepsilon}| p(n) = \Pr[X^n \in T_{n,\varepsilon}]$$

for $n \in \{1, \ldots, 1000\}$ and $\varepsilon = 0.1, 0.01$.

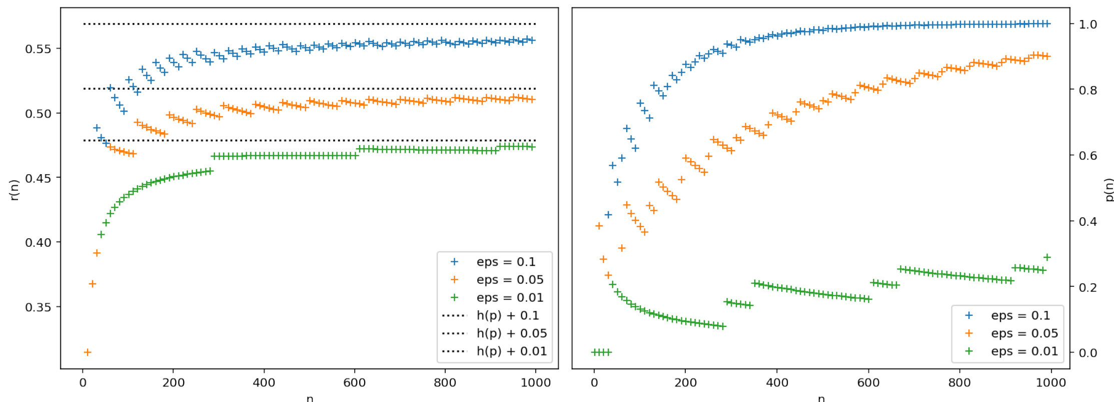
First compute the data:

[4]:
```
epsilons = [0.1, 0.05, 0.01]
ns = np.arange(1, 1001, 10)
rs = {}
ps = {}

ent = h(p)
for eps in epsilons:
    sizes = []
    probs = []
    for n in ns:
        # compute size and probability of typical subset
        size = 0
        prob = 0
        for k in range(1, n + 1):
            q = p**k * (1 - p)**(n - k)
            is_typical = 2**(-n * (ent + eps)) <= q <= 2**(-n * (ent - eps))
            d = binom(n, k)
            if is_typical:
                size += d
                prob += d * q
        sizes.append(size)
        probs.append(prob)
    # store r(n) and p(n) in dictionary
    with np.errstate(divide='ignore'):
        rs[eps] = np.log2(sizes) / ns
        ps[eps] = probs
```

Plot the functions:

[5]:
```
fig, axes = plt.subplots(ncols=2, figsize=(12.8, 4.8))
axes[0].set_xlabel('n')
axes[0].set_ylabel('r(n)')
axes[1].set_xlabel('n')
axes[1].set_ylabel('p(n)')
axes[1].yaxis.tick_right()
axes[1].yaxis.set_label_position('right')
```

```
for eps in rs:
    axes[0].plot(ns, rs[eps], '+', label='eps = %s' % eps)
    axes[0].hlines(
        h(p) + eps, 1, 1000, linestyle='dotted', label='h(p) + %s' % eps)
    axes[1].plot(ns, ps[eps], '+', label='eps = %s' % eps)
axes[0].legend()
axes[1].legend()
fig.tight_layout()
```



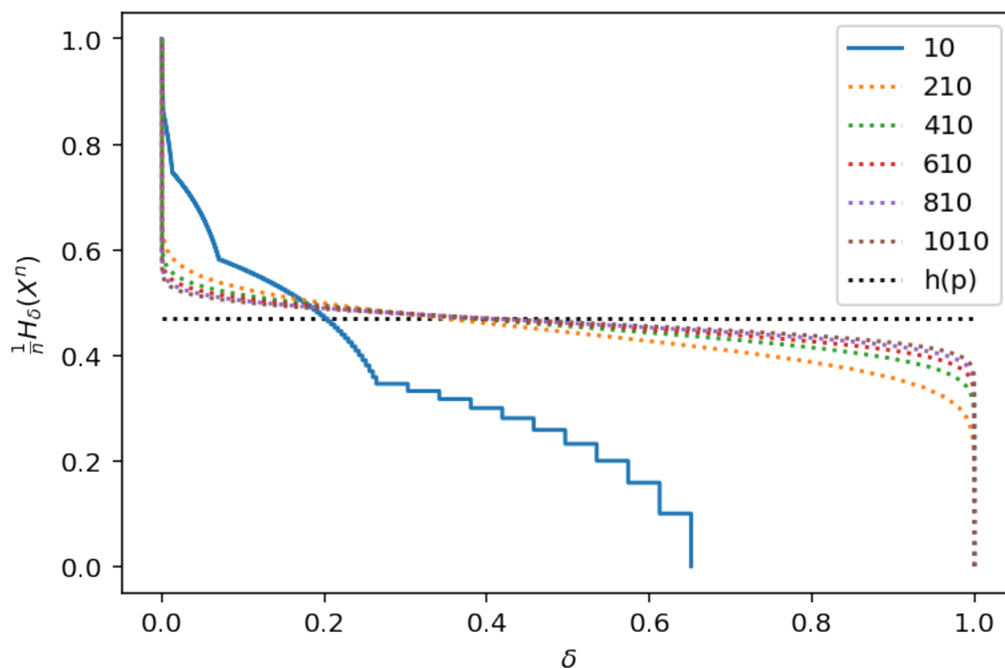
We see that $r(n)$ converges to $h(p) + \varepsilon$ and $p(n)$ converges to 1, as one would hope. Remark though that the convergence of $p(n)$ to 1 is very slow, especially if ε is small! Notice that since $|T_{n,\varepsilon}| \leq 2^{n(h(p)+\varepsilon)}$ we need $\lceil n(h(p) + \varepsilon) \rceil$ bits to send over elements of $T_{n,\varepsilon}$, so we can send information at rate $h(p) + \varepsilon$ with error probability as shown in the right hand figure. As you would expect, higher rates come with lower error probabilities!

However, we do not necessarily need to use the typical sets $T_{n,\varepsilon}$ for compression. An optimal choice of set S_δ to compress to with error δ is found by ordering all outcomes by probability, and then leaving out the outcomes with the smallest probabilities, until taking out one more outcome would lead to an error probability larger than δ. The essential bit content is then given by $H_\delta(X^n) = \log(|S_\delta|$. In our case, of a Bernoulli random variable with $p = 0.1$, the least likely sequence is the one which only has ones (it has probability 0.1^n). The next least likely sequences are the ones which have only a single zero. These have probability $0.9(0.1)^{n-1}$ and there are n of them.

In the following figure we will completely enumerate all outcomes for $n = 10$, and compute $H_\delta(X^n)$ and plot $\frac{1}{n}H_\delta(X^n)$ against δ. Next we will also do the same for larger n, but only compute the dependence of δ on k and interpolate (the set of all sequences is too large to enumerate over and we won't see the difference in the picture anyway). This reproduces figure 4.9 in MacKay. Make sure you understand the meaning of what is plotted (but don't worry about the code)!

[6]:
```
p = 0.1
plt.hlines(h(p), 0, 1, linestyle='dotted', label='h(p)')
# We completely enumerate for n = 10
n = 10
prob = np.zeros(2 ** n)
hs = np.log2(np.arange(1, 2 ** n + 1)) / n
```

```
i = 0
for k in np.arange(n + 1):
    for m in range(int(binom(n, k))):
        prob[i] = p**k * (1 - p)**(n - k)
        i += 1
deltas = 1 - np.cumsum(prob)
plt.step(deltas, hs, label=n)

# For larger n we interpolate by only looking at jumps in k
for n in [210, 410, 610, 810, 1010]:
    ks = np.arange(n + 1)
    ds = binom(n, ks)
    qs = ds * p**ks * (1 - p)**(n - ks)
    deltas = 1 - np.cumsum(qs)
    hs = np.log2(np.cumsum(ds))/n
    plt.plot(deltas, hs, linestyle='dotted', label=n)
    plt.xlabel(r'$\delta$')
    plt.ylabel(r'$\frac{1}{n}H_{\delta}(X^n)$')
    plt.legend()
```



We see that for large n the essential bit content satisfies $\frac{1}{n}H_\delta(X^n) \approx h(p)$ for δ away from 0 and 1 (and the larger n the better the approximation). This is precisely what Shannon's source coding theorem states! It means that for large n we can compress at rate approximately $h(p)$ for any nonzero allowed error. A nice exercise would be to make a version of this picture using the typical sets rather than the optimal sets and see how they differ!