**Mining Business-Topics in Source Language using Latent Dirichlet Allocation**

**• Abstract**

Preliminary results in some of the most current main-stream researches indicate that Latent Dirichlet Allocation (LDA) is able to identify some of the domain topics and is a satisfactory starting point for further manual refinement of topics.

Most of the large software maintenance efforts are tagged down by absence of documented business topics and source code,LDA serves to address this techniques of text mining .

# Categories and Subject Descriptors

Software Engineering and Machine Learning: Distribution, Maintenance, and Enhancement—Restructuring, reverse engineering, and reengineering;

**General terms**

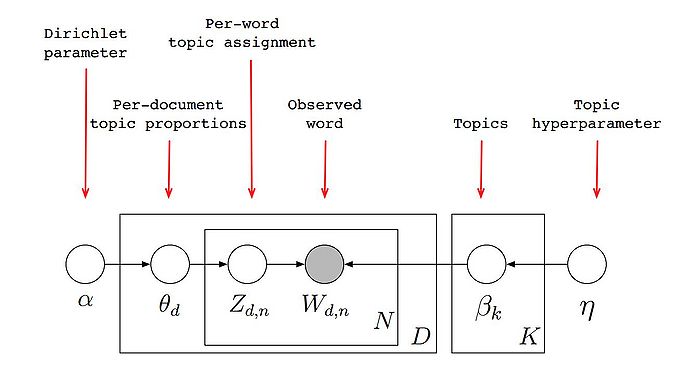
Latent Dirichlet Allocation, Domain Topic ,Business Domain Topic

**Keywords**

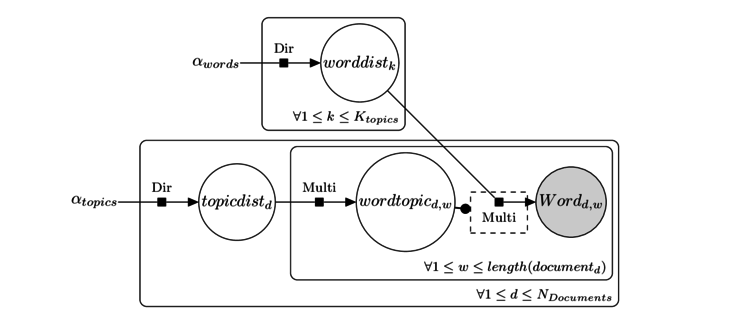
Maintenance of Software Systems, Program comprehension and LDA.

# INTRODUCTION

Large legacy software systems often exist in a state of disorganization with poor or no documentation. Adding new features and fixing bugs in such a system is highly error prone and time consuming since the original authors of the system.



Schematic Representations of LDA Algorithm



**Problem and Data** set(s) description (where you describe in detail the problem you want to solve and its significance)

**Methods**

**Experimental setup (including data pre-processing, feature selection and extraction)**

%**matplotlib** inline

# Sampling from a Hierarchical Dirichlet Process

[As we saw earlier](file:///E:\dirichlet-distribution\) the Dirichlet process describes the distribution of a random probability distribution. The Dirichlet process takes two parameters: a base distribution *H*0 and a dispersion parameter *α*. A sample from the Dirichlet process is itself a probability distribution that looks like *H*0. On average, the larger *α* is, the closer a sample from DP(*αH*0) will be to *H*0.

Suppose we're feeling masochistic and want to input a distribution sampled from a Dirichlet process as base distribution to a new Dirichlet process. (It will turn out that there are good reasons for this!) Conceptually this makes sense. But can we construct such a thing in practice? Said another way, can we build a sampler that will draw samples from a probability distribution drawn from these nested Dirichlet processes? We might initially try construct a sample (a probability distribution) from the first Dirichlet process before feeding it into the second.

But recall that fully constructing a sample (a probability distribution!) from a Dirichlet process would require drawing a countably infinite number of samples from *H*0 and from the beta distribution to generate the weights. This would take forever, even with Hadoop!

[Dan Roy, et al](http://danroy.org/papers/RoyManGooTen-ICMLNPB-2008.pdf) helpfully described a technique of using stochastic memoization to construct a distribution sampled from a Dirichlet process in a just-in-time manner. This process provides us with the equivalent of the [Scipy rvs](http://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.rv_continuous.rvs.html) method for the sampled distribution. Stochastic memoization is equivalent to the [Chinese restaurant process](http://www.cs.princeton.edu/courses/archive/fall07/cos597C/scribe/20070921.pdf): sometimes you get seated an an occupied table (i.e. sometimes you're given a sample you've seen before) and sometimes you're put at a new table (given a unique sample).

Here is our memoization class again:

In [162]:

**from** **numpy.random** **import** choice

**from** **scipy.stats** **import** beta

**class** **DirichletProcessSample**():

**def** \_\_init\_\_(self, base\_measure, alpha):

self.base\_measure = base\_measure

self.alpha = alpha

self.cache = []

self.weights = []

self.total\_stick\_used = 0.

**def** \_\_call\_\_(self):

remaining = 1.0 - self.total\_stick\_used

i = DirichletProcessSample.roll\_die(self.weights + [remaining])

**if** i **is** **not** **None** **and** i < len(self.weights) :

**return** self.cache[i]

**else**:

stick\_piece = beta(1, self.alpha).rvs() \* remaining

self.total\_stick\_used += stick\_piece

self.weights.append(stick\_piece)

new\_value = self.base\_measure()

self.cache.append(new\_value)

**return** new\_value

@staticmethod

**def** roll\_die(weights):

**if** weights:

**return** choice(range(len(weights)), p=weights)

**else**:

**return** **None**

Let's illustrate again with a standard normal base measure. We can construct a function base\_measure that generates samples from it.

In [95]:

**from** **scipy.stats** **import** norm

base\_measure = **lambda**: norm().rvs()

Because the normal distribution has continuous support, we can generate samples from it forever and we will never see the same sample twice (in theory). We can illustrate this by drawing from the distribution ten thousand times and seeing that we get ten thousand unique values.

In [163]:

**from** **pandas** **import** Series

ndraws = 10000

print("Number of unique samples after **{}** draws:".format(ndraws),)

draws = Series([base\_measure() **for** \_ **in** range(ndraws)])

print(draws.unique().size)

Number of unique samples after 10000 draws: 10000

However, when we feed the base measure through the stochastic memoization procedure and then sample, we get many duplicate samples. The number of unique samples goes down as *α* increases.

In [164]:

norm\_dp = DirichletProcessSample(base\_measure, alpha=100)

print("Number of unique samples after **{}** draws:".format(ndraws),)

dp\_draws = Series([norm\_dp() **for** \_ **in** range(ndraws)])

print(dp\_draws.unique().size)

Number of unique samples after 10000 draws: 446

At this point, we have a function dp\_draws that returns samples from a probability distribution (specifically, a probability distribution sampled from DP(*αH*0)). We can use dp\_draws as a base distribution for another Dirichlet process!

In [155]:

norm\_hdp = DirichletProcessSample(norm\_dp, alpha=10)

How do we interpret this? norm\_dp is a sampler from a probability distribution that looks like the standard normal distribution. norm\_hdp is a sampler from a probability distribution that "looks like" the distribution norm\_dp samples from.

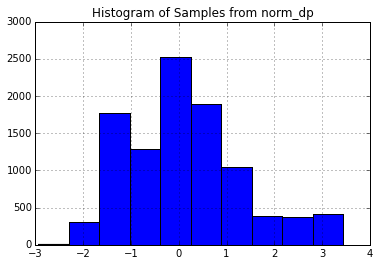
Here is a histogram of samples drawn from norm\_dp, our first sampler.

In [152]:

**import** **matplotlib.pyplot** **as** **plt**

Series(norm\_dp() **for** \_ **in** range(10000)).hist()

\_=plt.title("Histogram of Samples from norm\_dp")

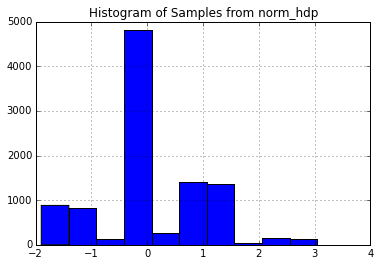
****

And here is a histogram for samples drawn from norm\_hdp, our second sampler.

In [154]:

Series(norm\_hdp() **for** \_ **in** range(10000)).hist()

\_=plt.title("Histogram of Samples from norm\_hdp")

****

The second plot doesn't look very much like the first! The level to which a sample from a Dirichlet process approximates the base distribution is a function of the dispersion parameter *α*. Because I set *α*=10 (which is relatively small), the approximation is fairly course. In terms of memoization, a small *α* value means the stochastic memoizer will more frequently reuse values already seen instead of drawing new ones.

This nesting procedure, where a sample from one Dirichlet process is fed into another Dirichlet process as a base distribution, is more than just a curiousity. It is known as a [Hierarchical Dirichlet Process, and it plays an important role in the study of Bayesian Nonparametrics](http://www.cs.berkeley.edu/~jordan/papers/hdp.pdf).

Without the stochastic memoization framework, constructing a sampler for a hierarchical Dirichlet process is a daunting task. We want to be able to draw samples from a distribution drawn from the second level Dirichlet process. However, to be able to do that, we need to be able to draw samples from a distribution sampled from a base distribution of the second-level Dirichlet process: this base distribution is a distribution drawn from the first-level Dirichlet process.

Though it appeared that we would need to be able to fully construct the first level sample (by drawing a countably infinite number of samples from the first-level base distribution). However, stochastic memoization allows us to only construct the first distribution just-in-time as it is needed at the second-level.

We can define a Python class to encapsulate the Hierarchical Dirichlet Process as a base class of the Dirichlet process.

In [165]:

**class** **HierarchicalDirichletProcessSample**(DirichletProcessSample):

**def** \_\_init\_\_(self, base\_measure, alpha1, alpha2):

first\_level\_dp = DirichletProcessSample(base\_measure, alpha1)

self.second\_level\_dp = DirichletProcessSample(first\_level\_dp, alpha2)

**def** \_\_call\_\_(self):

**return** self.second\_level\_dp()

Since the Hierarchical DP is a Dirichlet Process inside of Dirichlet process, we must provide it with both a first and second level *α* value.

In [167]:

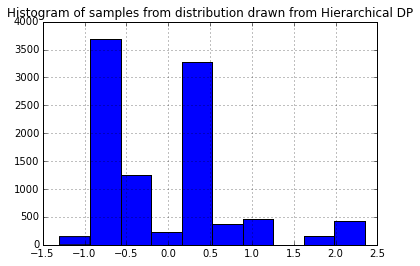
norm\_hdp = HierarchicalDirichletProcessSample(base\_measure, alpha1=10, alpha2=20)

We can sample directly from the probability distribution drawn from the Hierarchical Dirichlet Process.

In [170]:

Series(norm\_hdp() **for** \_ **in** range(10000)).hist()

\_=plt.title("Histogram of samples from distribution drawn from Hierarchical DP")

****

norm\_hdp is not equivalent to the Hierarchical Dirichlet Process; it samples from a single distribution sampled from this HDP. Each time we instantiate the norm\_hdp variable, we are getting a sampler for a unique distribution. Below we sample five times and get five different distributions.

In [180]:

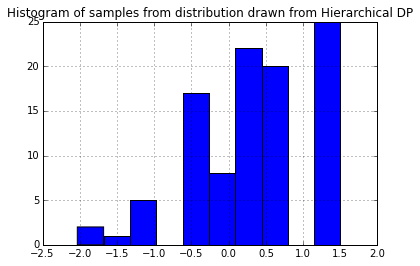
**for** i **in** range(5):

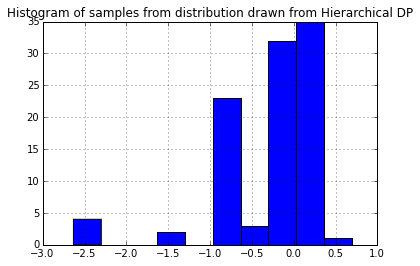
norm\_hdp = HierarchicalDirichletProcessSample(base\_measure, alpha1=10, alpha2=10)

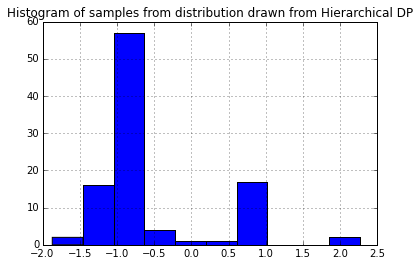
\_=Series(norm\_hdp() **for** \_ **in** range(100)).hist()

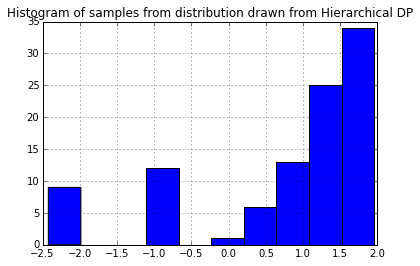
\_=plt.title("Histogram of samples from distribution drawn from Hierarchical DP")

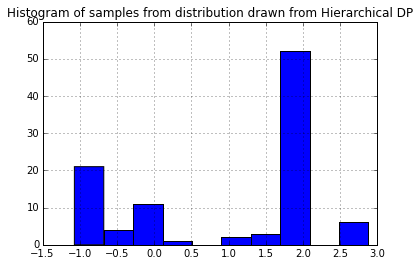
\_=plt.figure()

****

****

****

****

****

<matplotlib.figure.Figure at 0x112a2da50>

In a later post, I will discuss how these tools are applied in the realm of Bayesian nonparametrics.

**Results**

%**matplotlib** inline

## Dirichlet Distribution

The symmetric [Dirichlet distribution](https://en.wikipedia.org/wiki/Dirichlet_distribution) (DD) can be considered a distribution of distributions. Each sample from the DD is a [categorial distribution](https://en.wikipedia.org/wiki/Categorical_distribution) over KK categories. It is parameterized G0G0, a distribution over KK categories and αα, a scale factor.

The expected value of the DD is G0G0. The variance of the DD is a function of the scale factor. When αα is large, samples from DD(α⋅G0)DD(α⋅G0) will be very close to G0G0. When αα is small, samples will vary more widely.

We demonstrate below by setting G0=[.2,.2,.6]G0=[.2,.2,.6] and varying αα from 0.1 to 1000. In each case, the mean of the samples is roughly G0G0, but the standard deviation is decreases as αα increases.

In [10]:

**import** **numpy** **as** **np**

**from** **scipy.stats** **import** dirichlet

np.set\_printoptions(precision=2)

**def** stats(scale\_factor, G0=[.2, .2, .6], N=10000):

samples = dirichlet(alpha = scale\_factor \* np.array(G0)).rvs(N)

print(" alpha:", scale\_factor)

print(" element-wise mean:", samples.mean(axis=0))

print("element-wise standard deviation:", samples.std(axis=0))

print()

**for** scale **in** [0.1, 1, 10, 100, 1000]:

stats(scale)

alpha: 0.1

element-wise mean: [ 0.2 0.2 0.6]

element-wise standard deviation: [ 0.38 0.38 0.47]

alpha: 1

element-wise mean: [ 0.2 0.2 0.6]

element-wise standard deviation: [ 0.28 0.28 0.35]

alpha: 10

element-wise mean: [ 0.2 0.2 0.6]

element-wise standard deviation: [ 0.12 0.12 0.15]

alpha: 100

element-wise mean: [ 0.2 0.2 0.6]

element-wise standard deviation: [ 0.04 0.04 0.05]

alpha: 1000

element-wise mean: [ 0.2 0.2 0.6]

element-wise standard deviation: [ 0.01 0.01 0.02]

## Dirichlet Process

The [Dirichlet Process](https://en.wikipedia.org/wiki/Dirichlet_process) can be considered a way to generalize the Dirichlet distribution. While the Dirichlet distribution is parameterized by a discrete distribution G0G0 and generates samples that are similar discrete distributions, the Dirichlet process is parameterized by a generic distribution H0H0 and generates samples which are distributions similar to H0H0. The Dirichlet process also has a parameter αα that determines how similar how widely samples will vary from H0H0.

We can construct a sample HH (recall that HH is a probability distribution) from a Dirichlet process DP(αH0)DP(αH0) by drawing a countably infinite number of samples θkθk from H0H0) and setting:

H=∑k=1∞πk⋅δ(x−θk)H=∑k=1∞πk⋅δ(x−θk)

where the πkπk are carefully chosen weights (more later) that sum to 1. (δδ is the [Dirac delta function](https://en.wikipedia.org/wiki/Dirac_delta_function).)

HH, a sample from DP(αH0)DP(αH0), is a probability distribution that looks similar to H0H0 (also a distribution). In particular, HH is a discrete distribution that takes the value θkθk with probability πkπk. This sampled distribution HH is a discrete distribution \_even if H0H0 has continuous support\_; the [support](http://www.statlect.com/glossary/support_of_a_random_variable.htm) of HH is a countably infinite subset of the support H0H0.

The weights (πkπk values) of a Dirichlet process sample related the Dirichlet process back to the Dirichlet distribution.

[Gregor Heinrich](http://www.arbylon.net/publications/ilda.pdf) writes:

The defining property of the DP is that its samples have weights πkπk and locations θkθk distributed in such a way that when partitioning S(H)S(H) into finitely many arbitrary disjoint subsets S1,…,SjS1,…,Sj J<∞J<∞, the sums of the weights πkπk in each of these JJ subsets are distributed according to a Dirichlet distribution that is parameterized by αα and a discrete base distribution (like G0G0) whose weights are equal to the integrals of the base distribution H0H0 over the subsets SnSn.

As an example, Heinrich imagines a DP with a standard normal base measure H0∼N(0,1)H0∼N(0,1). Let HH be a sample from DP(H0)DP(H0) and partition the real line (the support of a normal distribution) as S1=(−∞,−1]S1=(−∞,−1], S2=(−1,1]S2=(−1,1], and S3=(1,∞]S3=(1,∞] then

H(S1),H(S2),H(S3)∼Dir(αerf(−1),α(erf(1)−erf(−1)),α(1−erf(1)))H(S1),H(S2),H(S3)∼Dir(αerf(−1),α(erf(1)−erf(−1)),α(1−erf(1)))

where H(Sn)H(Sn) be the sum of the πkπk values whose θkθk lie in SnSn.

These SnSn subsets are chosen for convenience, however similar results would hold for any choice of SnSn. For any sample from a Dirichlet process, we can construct a sample from a Dirichlet distribution by partitioning the support of the sample into a finite number of bins.

There are several equivalent ways to choose the πkπk so that this property is satisfied: the Chinese restaurant process, the stick-breaking process, and the Pólya urn scheme.

To generate {πk}{πk} according to a stick-breaking process we define βkβk to be a sample from Beta(1,α)Beta(1,α). π1π1 is equal to β1β1. Successive values are defined recursively as

πk=βk∏j=1k−1(1−βj).πk=βk∏j=1k−1(1−βj).

Thus, if we want to draw a sample from a Dirichlet process, we could, in theory, sample an infinite number of θkθk values from the base distribution H0H0, an infinite number of βkβk values from the Beta distribution. Of course, sampling an infinite number of values is easier in theory than in practice.

However, by noting that the πkπk values are positive values summing to 1, we note that, in expectation, they must get increasingly small as k→∞k→∞. Thus, we can reasonably approximate a sample H∼DP(αH0)H∼DP(αH0) by drawing enough samples such that ∑Kk=1πk≈1∑k=1Kπk≈1.

We use this method below to draw approximate samples from several Dirichlet processes with a standard normal (N(0,1)N(0,1)) base distribution but varying αα values.

Recall that a single sample from a Dirichlet process is a probability distribution over a countably infinite subset of the support of the base measure.

The blue line is the PDF for a standard normal. The black lines represent the θkθk and πkπk values; θkθk is indicated by the position of the black line on the xx-axis; πkπk is proportional to the height of each line.

We generate enough πkπk values are generated so their sum is greater than 0.99. When αα is small, very few θkθk's will have corresponding πkπk values larger than 0.010.01. However, as αα grows large, the sample becomes a more accurate (though still discrete) approximation of N(0,1)N(0,1).

In [13]:

**import** **matplotlib.pyplot** **as** **plt**

**from** **scipy.stats** **import** beta, norm

**def** dirichlet\_sample\_approximation(base\_measure, alpha, tol=0.01):

betas = []

pis = []

betas.append(beta(1, alpha).rvs())

pis.append(betas[0])

**while** sum(pis) < (1.-tol):

s = np.sum([np.log(1 - b) **for** b **in** betas])

new\_beta = beta(1, alpha).rvs()

betas.append(new\_beta)

pis.append(new\_beta \* np.exp(s))

pis = np.array(pis)

thetas = np.array([base\_measure() **for** \_ **in** pis])

**return** pis, thetas

**def** plot\_normal\_dp\_approximation(alpha):

plt.figure()

plt.title("Dirichlet Process Sample with N(0,1) Base Measure")

plt.suptitle("alpha: **%s**" % alpha)

pis, thetas = dirichlet\_sample\_approximation(**lambda**: norm().rvs(), alpha)

pis = pis \* (norm.pdf(0) / pis.max())

plt.vlines(thetas, 0, pis, )

X = np.linspace(-4,4,100)

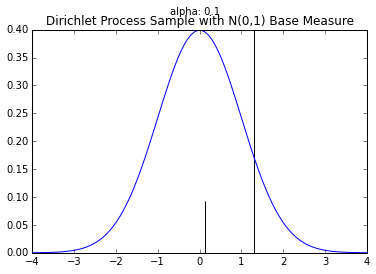
plt.plot(X, norm.pdf(X))

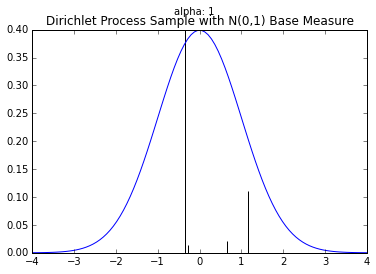
plot\_normal\_dp\_approximation(.1)

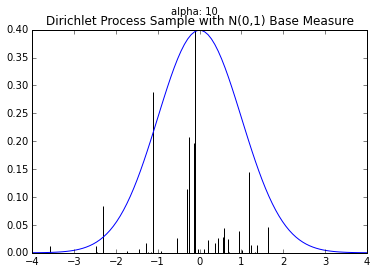
plot\_normal\_dp\_approximation(1)

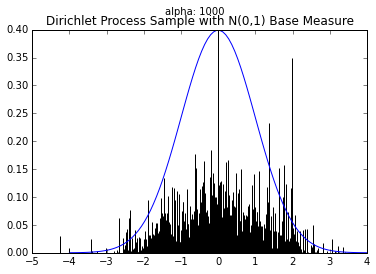
plot\_normal\_dp\_approximation(10)

plot\_normal\_dp\_approximation(1000)

****

****

****

****

Often we want to draw samples from a distribution sampled from a Dirichlet process instead of from the Dirichlet process itself. Much of the literature on the topic unhelpful refers to this as sampling from a Dirichlet process.

Fortunately, we don't have to draw an infinite number of samples from the base distribution and stick breaking process to do this. Instead, we can draw these samples as they are needed.

Suppose, for example, we know a finite number of the θkθk and πkπk values for a sample H∼Dir(αH0)H∼Dir(αH0). For example, we know

π1=0.5,π2=0.3,θ1=0.1,θ2=−0.5.π1=0.5,π2=0.3,θ1=0.1,θ2=−0.5.

To sample from HH, we can generate a uniform random uu number between 0 and 1. If uu is less than 0.5, our sample is 0.10.1. If 0.5<=u<0.80.5<=u<0.8, our sample is −0.5−0.5. If u>=0.8u>=0.8, our sample (from HH will be a new sample θ3θ3 from H0H0. At the same time, we should also sample and store π3π3. When we draw our next sample, we will again draw u∼Uniform(0,1)u∼Uniform(0,1) but will compare against π1,π2π1,π2, AND π3π3.

The class below will take a base distribution H0H0 and αα as arguments to its constructor. The class instance can then be called to generate samples from H∼DP(αH0)H∼DP(αH0).

In [20]:

**from** **numpy.random** **import** choice

**class** **DirichletProcessSample**():

**def** \_\_init\_\_(self, base\_measure, alpha):

self.base\_measure = base\_measure

self.alpha = alpha

self.cache = []

self.weights = []

self.total\_stick\_used = 0.

**def** \_\_call\_\_(self):

remaining = 1.0 - self.total\_stick\_used

i = DirichletProcessSample.roll\_die(self.weights + [remaining])

**if** i **is** **not** **None** **and** i < len(self.weights) :

**return** self.cache[i]

**else**:

stick\_piece = beta(1, self.alpha).rvs() \* remaining

self.total\_stick\_used += stick\_piece

self.weights.append(stick\_piece)

new\_value = self.base\_measure()

self.cache.append(new\_value)

**return** new\_value

@staticmethod

**def** roll\_die(weights):

**if** weights:

**return** choice(range(len(weights)), p=weights)

**else**:

**return** **None**

This Dirichlet process class could be called stochastic memoization. This idea was first articulated in somewhat abstruse terms by [Daniel Roy, et al](http://danroy.org/papers/RoyManGooTen-ICMLNPB-2008.pdf).

Below are histograms of 10000 samples drawn from samples drawn from Dirichlet processes with standard normal base distribution and varying αα values.

In [22]:

**import** **pandas** **as** **pd**

base\_measure = **lambda**: norm().rvs()

n\_samples = 10000

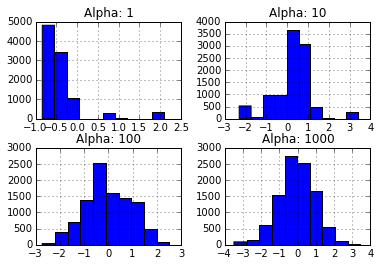
samples = {}

**for** alpha **in** [1, 10, 100, 1000]:

dirichlet\_norm = DirichletProcessSample(base\_measure=base\_measure, alpha=alpha)

samples["Alpha: **%s**" % alpha] = [dirichlet\_norm() **for** \_ **in** range(n\_samples)]

\_ = pd.DataFrame(samples).hist()

****

Note that these histograms look very similar to the corresponding plots of sampled distributions above. However, these histograms are plotting points sampled from a distribution sampled from a Dirichlet process while the plots above were showing approximate distributions samples from the Dirichlet process. Of course, as the number of samples from each HH grows large, we would expect the histogram to be a very good empirical approximation of HH.

[In a another post](file:///E:\hdp-sampling\), I will look at how this DirichletProcessSample class can be used to draw samples from a hierarchical Dirichlet process.

In [ ]:

**Discussion and Conclusions**

**References**

# REFERENCES

1. Source navigator 5.4.1. http://sourcenav.sourceforge.net, 2003.
2. P. Anderson and M. Zarins. The codesurfer software understanding platform. In *IWPC*, pages 147–148. IEEE Computer Society, 2005.
3. N. Anquetil and T. C. Lethbridge. Recovering software architecture from the names of source files. *Journal of Software Maintenance: Research and Practice*, 11:201–221, 1999.
4. G. Antoniol, G. Canfora, G. Casazza, A. D. Lucia, and E. Merlo. Recovering traceability links between code and documentation. *IEEE Transactions in Software Engineering*, 28(10):970–983, 2002.
5. T. J. Biggerstaff, B. G. Mitbander, and D. Webster. Program understanding and the concept assignment problem. *Communications of the ACM*, 37(5):72–83, May 1994.
6. Z. Bin, M. David, and L. Xinghua. Identifying biological concepts from a protein-related corpus with a probabilistic topic model. *BMC Bioinformatics*, 7, 2006.
7. D. Blei, A. Ng, and M. Jordan. Latent dirichlet allocation. *Journal of Machine Learning Research*, 3:993–1022, 2003.
8. B. Caprile and P. Tonella. Nomen est omen:

Analyzing the language of function identifiers. In *Proceedings of the Sixth Working Conference on Reverse Engineering*, 1999.

1. S. Deerwester, S. T. Dumais, G. W. Furnas, and T. K. Landauer. Indexing by latent semantic analysis. *Journal of the American Society for Information Science*, 41:391–407, 1990.
2. T. Griffiths and M. Steyvers. Finding scientific topics. In *Proceedings of the National Academy of Sciences*, pages 5228–5235, 2004.
3. S. Kawaguchi, P. K. Garg, M. Matsushita, and K. Inoue. MUDABlue: An automatic categorization system for open source repositories. In *APSEC*, pages 184–193. IEEE Computer Society, 2004.
4. A. Kuhn. Semantic clustering: Making use of linguistic information to reveal concepts in source code. Master’s thesis, University of Bern, 2006.
5. A. Kuhn, S. Ducasse, and T. Gˆırba. Semantic clustering: Identifying topics in source code. *IST*, 2006. To appear.
6. J. Lafferty and T. Minka. Expectation-propagation for the generative aspect model. In *Proceedings of the 18th Conference on Uncertainty in Artificial Intelligence*, 2002.
7. A. Marcus and J. I. Maletic. Identification of high-level concept clones in source code. In *Proceedings of the 16th International Conference on Automated Software Engineering (ASE 2001)*, pages 107–114, Nov. 2001.
8. A. Marcus and J. I. Maletic. Recovering documentation-to-source-code traceability links using latent semantic indexing. In *International Conference on Software Engineering*, pages 125–134. IEEE Computer Society Press, may 2003.
9. A. Marcus, A. Sergeyev, V. Rajlich, and J. Maletic. An information retrieval approach to concept location in source code. In *Proceedings of the 11th Working Conference on Reverse Engineering (WCRE 2004)*, pages 214–223, Nov. 2004.
10. A. McCallum, A. Corrada-Emmanuel, and X. Wang. Topic and role discovery in social networks. In L. P. Kaelbling and A. Saffiotti, editors, *IJCAI*, pages 786–791. Professional Book Center, 2005.
11. D. Newman, C. Chemudugunta, P. Smyth, and M. Steyvers. Analyzing entities and topics in news articles using statistical topic models. In *Lecture Notes on Computer Science*. Springer-Verlag, 2006.