Progress report

For the `SIMULATE, ESTIMATE, INFER` stuff, I have made a simple Python interface for each of the data type specifications ('numerical', 'constrained' and 'discrete') using the simplifying assumption of Naive Bayes, that is, that all columns are independent. This assumption carries direct consequences over the three operations:

- `ESTIMATE` returns 0 for every pair of columns
- `SIMULATE` returns samples from the (marginal) joint predictive distribution
- `INFER` returns an estimator from the joint predictive distribution for each NaN in the data

Additionally, I created methods for computing the logpdf for the joint predictive distribution (not tested yet). For the logpdf of the constrained numerical data, I normalized the pdf of the non-constrained numerical in the constrained range.

For `INFER`, the chosen estimator was

$$\hat{x} = \arg\max_x P(x|D).$$

For numerical variables, this is the predictive mean (if the mean lies in the model's domain). For categorical variables, the confidence was then $P(\hat{x}|D)$. For numerical variables, the provisory confidence was set to 1, due to a *lack of a better heuristic*.

I started working on the mixture model with k-means, but I am still looking for a good metric for choosing the number of clusters to be decided through cross-validation. Sum of distances to centroids seems to always favor a larger number of clusters.

Questions:

- What is a proper metric for k-means cross-validation? Inertia (sum of distances to centroids) apparently does not work.
- What is a good confidence metric for `INFER` in continuous variables? It should depend on the variance of the data, or on the probability density of the point, but I have not yet decided what would be best.
- How to check asymptotic consistency in the unrealizable case? In other words, how to check that the predictive distribution is the one closest to the ground truth, given the hypothesis space? I thought of checking whether the predictive mean and variance converge to the sample mean and variance, at least in the numerical case, but I have no guarantee that this is the proper test.

Disclaimer

Since task **one** is taking me longer than I expected, I have decided to focus my efforts on it to get results as quickly as I can. Therefore, I have not worked on task **two** during the past week. One bottleneck for my progress was a proper understanding of the assumptions involved in this task, which happened last Wednesday after talking to Feras. Most of my current progress started then.

Plotting examples to test Metamodels

In [1]:
```
%matplotlib inline
import numpy as np
import seaborn as sns
from matplotlib import pyplot
from NIGNormalMetamodel import *
from ConstrainedNIGNormalMetamodel import *
from CategoricalMetamodel import *
```

Provisory:

I have first implemented Naive Bayes for the different data types separately. In a future version, I will add them together in a `generator_emulator` class, which identifies data type per column. For now, the `generator_emulator` calls one model for each data type.

In [2]:
```
def generator_emulator(data, typespec, hypers=[], bounds=[]):
    if typespec == 'numerical':
        return NIGNormalMetamodel(data,typespec, hypers)
    elif typespec == 'constrained':
        return ConstrainedNIGNormalMetamodel(data,typespec,hypers,bounds)
    elif typespec == 'discrete':
        return CategoricalMetamodel(data,typespec,hypers)
    else:
        raise TypeError('Possible types are `numerical`, `constrained`, and`di
screte`')
```

Heuristics:

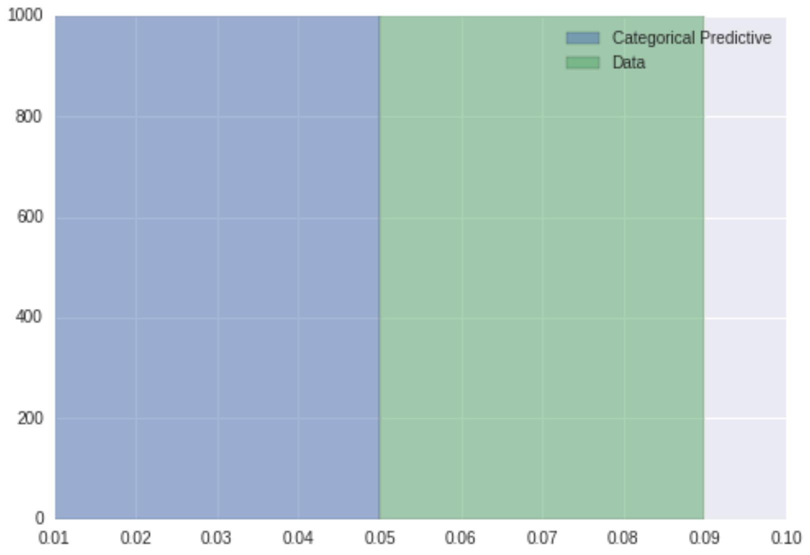
Test all models using different data generating processes. In each of them, plot (carefully labeled) graphs showing that:

- The sampler (`simulate`) converges to the data distribution.
- The model is correctly specified
  - Predictive `logpdf` and samples match, as well as mean and variance
  - Modes of the posterior distribution are close to the true parameters

Single-valued categorical

```
In [3]: # Generate data
        X = np.tile(0., [1000,1])

        # Sample from predictive distribution
        discrete = generator_emulator(X,'discrete')
        _,Y_disc = discrete.simulate([0],numpredictions=1000)

        # Reshape sampled data to ndarray
        Y_disc = (Y_disc[:][None]).transpose()
        data = np.append(X, Y_disc,axis=1)

        # Plot original data and predicted samples
        pyplot.hist(data, alpha=0.5, label={'Data','Categorical Predictive'})

        pyplot.legend()
```

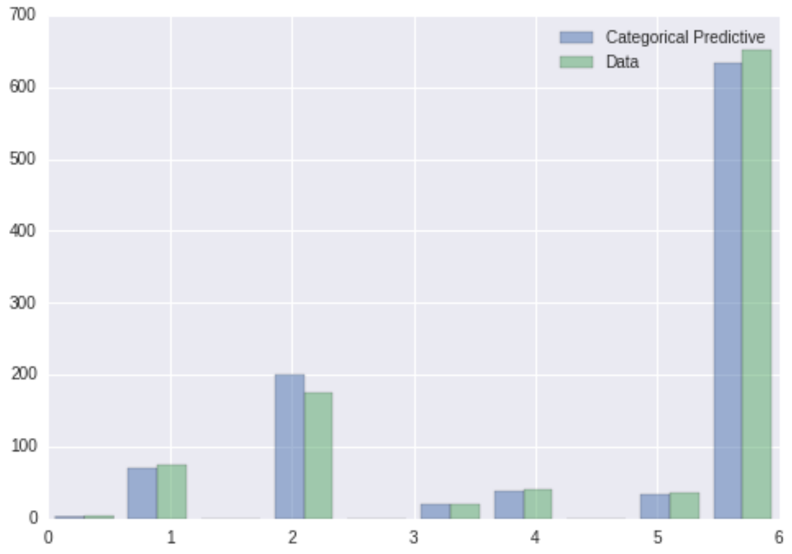
```
Out[3]: <matplotlib.legend.Legend at 0x7efed57f6d90>
```



Categorical with K = 7

```
In [6]: # Generate data
        counts = np.array(np.random.multinomial(1000, np.random.dirichlet(np.ones(7)/2
        )), dtype=int)
        X = counts_to_data(counts)

        # Sample from predictive distribution
        discrete = generator_emulator(X,'discrete')
        _,Y_disc = discrete.simulate([0],numpredictions=1000)

        # Reshape sampled data to ndarray
        Y_disc = (Y_disc[:][None]).transpose()
        data = np.append(X, Y_disc,axis=1)

        # Plot original data and predicted samples
        pyplot.hist(data, alpha=0.5, label={'Data','Categorical Predictive'})

        pyplot.legend()
```

```
Out[6]: <matplotlib.legend.Legend at 0x7f9f876ade10>
```

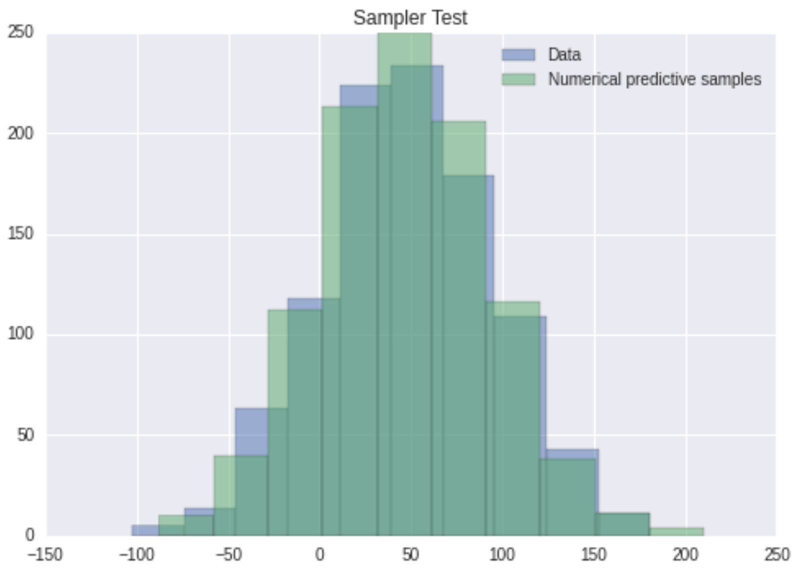


Normal

$$\mu = 50, \sigma = 50$$

```
In [33]: # Generate data
         X = np.random.normal(loc=50, scale=50.00,  size=[1000,1])

         # Sample from predictive distribution
         numerical = generator_emulator(X,'numerical')
         _,Y_num = numerical.simulate([0],numpredictions=1000)

         # Plot original data and predicted samples
         pyplot.hist(X, alpha=0.5, label='Data')
         pyplot.hist(Y_num, alpha=0.5, label='Numerical predictive samples')

         pyplot.title('Sampler Test')
         pyplot.legend()
```

```
Out[33]: <matplotlib.legend.Legend at 0x7efed0244dd0>
```

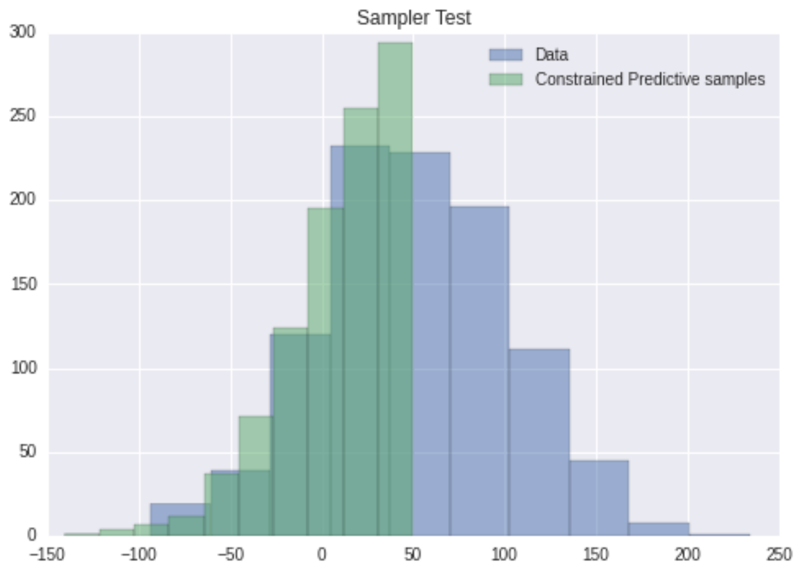


Normal data, constrained normal model

```
In [32]: constrained = generator_emulator(X,'constrained', bounds = [float('-inf'),50])
         _,Y_cons = constrained.simulate([0],numpredictions=1000)

         pyplot.hist(X, alpha=0.5, label='Data')
         pyplot.hist(Y_cons, alpha=0.5, label='Constrained predictive samples')

         pyplot.title('Sampler Test')
         pyplot.legend()
```

```
Constraints check was not implemented yet
```

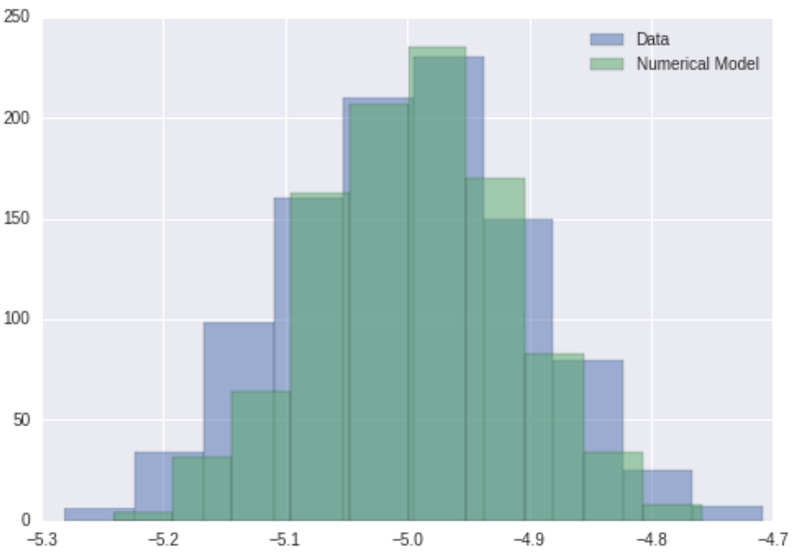
```
Out[32]: <matplotlib.legend.Legend at 0x7efed031db10>
```



Small variance

$$\mu = -5, \sigma = 0.1$$

```
In [14]: # Generate data
         X = np.random.normal(loc=-5, scale=0.1,  size=[1000,1])

         # Sample from predictive distribution
         numerical = generator_emulator(X,'numerical')
         _,Y_num = numerical.simulate([0],numpredictions=1000)

         # Plot original data and predicted samples
         pyplot.hist(X, alpha=0.5, label='Data')
         pyplot.hist(Y_num, alpha=0.5, label='Numerical Model')

         pyplot.legend()

Out[14]: <matplotlib.legend.Legend at 0x7efed4e44f50>
```

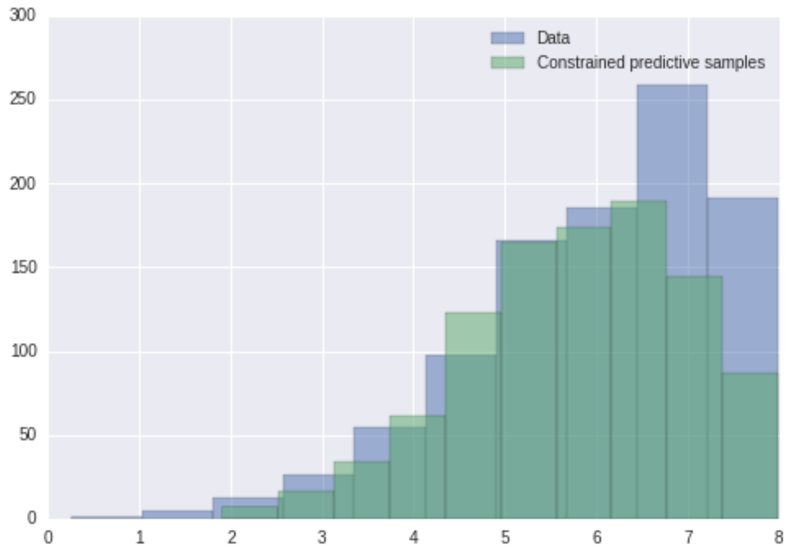


Constrained Normal

In [35]:
```
# Generate data
lower_bound, upper_bound = [0,8]
X = np.array([])
for ipred in np.arange(1000):
    in_range = False
    while not(in_range): # Reject samples out of range
        sample = np.random.normal(loc=7.0, scale=2.0, size=[1,1])
        if lower_bound <= sample and sample <= upper_bound:
            in_range = True
    X = np.append(X, sample)
X = X[:][np.newaxis].transpose()

# Sample from predictive distribution
numerical = generator_emulator(X,'constrained', bounds=[0,8])
_,Y_num = numerical.simulate([0],numpredictions=1000)

# Plot original data and predicted samples
pyplot.hist(X, alpha=0.5, label='Data')
pyplot.hist(Y_num, alpha=0.5, label='Constrained predictive samples')

pyplot.legend()
```

```
Constraints check was not implemented yet
```

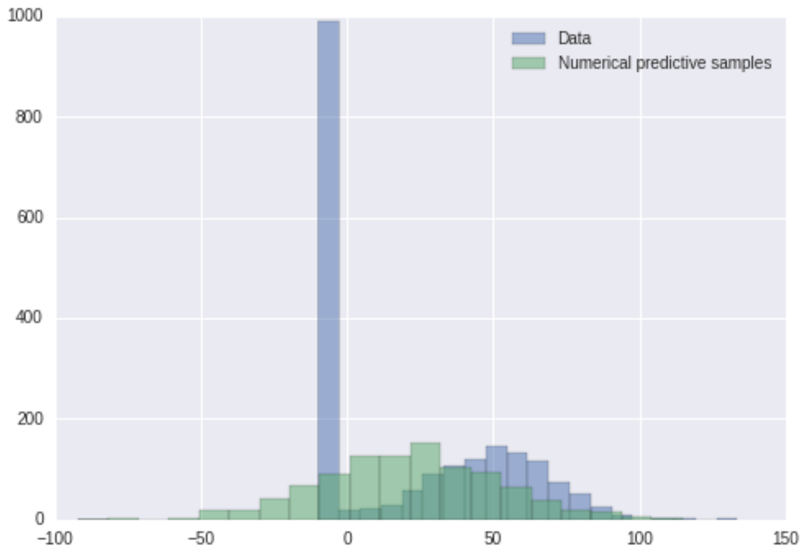
Out[35]:
```
<matplotlib.legend.Legend at 0x7efecfd64e50>
```



Mixture of Normals

$$\mu_1 = -5, \sigma_1 = 1$$
$$\mu_2 = 50, \sigma_2 = 20$$

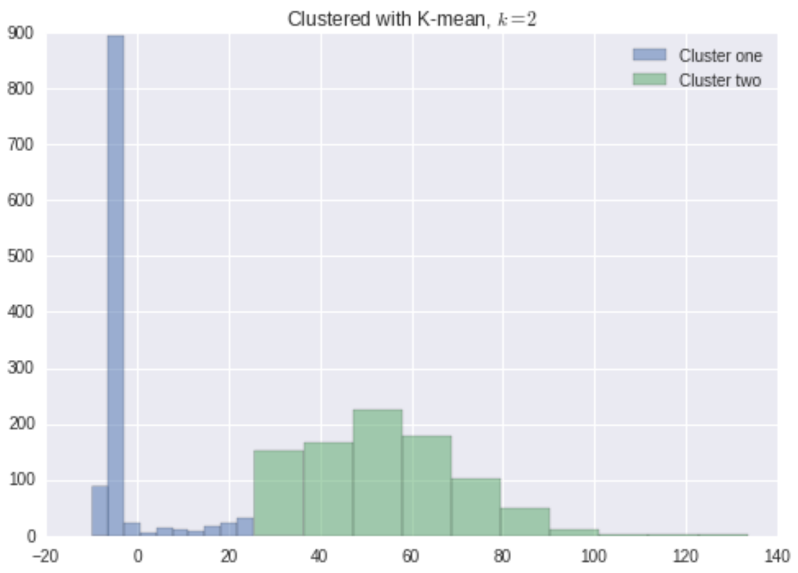
In [38]:
```
X = np.random.normal(loc=-5, scale=1,  size=[1000,1])
X = np.append(X, np.random.normal(loc=50, scale=20, size = [1000,1]), axis=0)
# X = np.append(X, np.random.normal(loc=50, scale=10, size = [1000,1]), axis=0
)

numerical = generator_emulator(X,'numerical')
_,Y_num = numerical.simulate([0],numpredictions=1000)

pyplot.hist(X, bins=20, alpha=0.5,label='Data')
pyplot.hist(Y_num, bins=20, alpha=0.5, label='Numerical predictive samples')

pyplot.legend()
pyplot.show()
```



After K-means clustering (unfinished)

```
In [44]: from sklearn.cluster import KMeans

         est = KMeans(n_clusters=2)
         est.fit(X)
         labels = est.labels_
         centers = est.cluster_centers_
         inertia = est.inertia_

         pyplot.hist(X[labels==0], alpha=0.5, label='Cluster one')
         pyplot.hist(X[labels==1], alpha=0.5, label='Cluster two')

         pyplot.legend()
         pyplot.title('Clustered with K-mean, $k=2$')
```

```
Out[44]: <matplotlib.text.Text at 0x7efecf898f50>
```

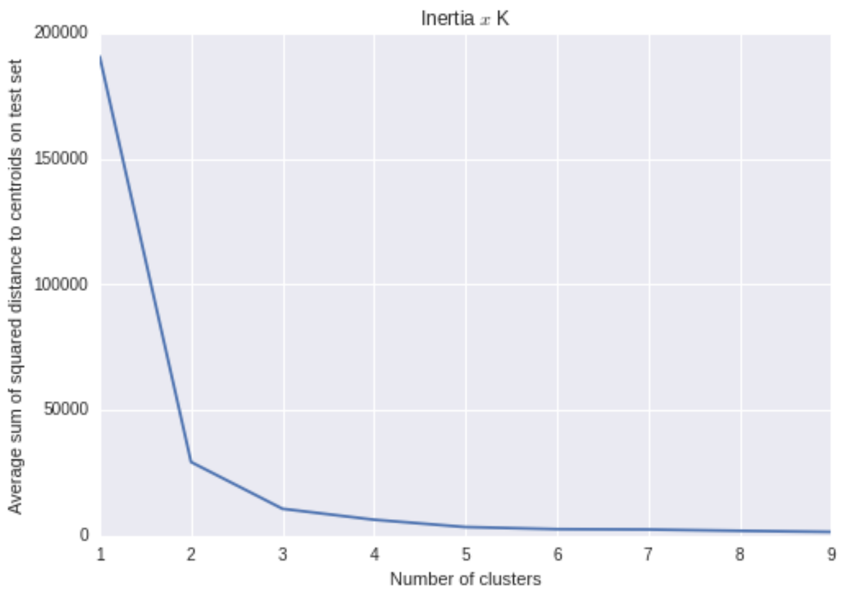


K-Means with cross-validation

- Sum of distances to centroid does not work as metric

In [48]:
```
from sklearn.cluster import KMeans
from sklearn import cross_validation

def get_inertia(data, labels, centers):
    inertia = 0
    for i_label in np.unique(labels):
        distances = data[labels==i_label] - centers[i_label]
        inertia += sum(distances**2)
    return inertia

def KMeansCV(X, k_range=np.arange(1,10), test_size=0.1):

    all_centers = []
    inertia = []
    for i_clusters in k_range:

        cluster_inertia = 0
        no_splits = int(1/test_size)
        for i_split in np.arange(no_splits):
            X_train, X_test = cross_validation.train_test_split(X, test_size=t
est_size)

            est = KMeans(n_clusters=i_clusters)
            est.fit(X_train)

            centers = est.cluster_centers_
            predicted_labels = est.predict(X_test)

            cluster_inertia += get_inertia(X_test, predicted_labels, centers)
/ no_splits

        all_centers.append(centers)
        inertia.append(cluster_inertia)

    return inertia, all_centers
```

```
In [54]: inertia, allcenters = KMeansCV(X)
         pyplot.plot(np.arange(1,10),inertia)

         pyplot.title('Inertia $x$ K ')
         pyplot.xlabel('Number of clusters')
         pyplot.ylabel('Average sum of squared distance to centroids on test set')

Out[54]: <matplotlib.text.Text at 0x7efecf4d8f10>
```


Inertia x K
Average sum of squared distance to centroids on test set
Number of clusters