Homework - Bayesian modeling - Part C (30 points)

Implementing the Metropolis–Hastings algorithm for a Bayesian model of speech perception

by *Brenden Lake* and *Todd Gureckis*
Computational Cognitive Modeling
NYU class webpage: https://brendenlake.github.io/CCM-site/ (https://brendenlake.github.io/CCM-site/)

This homework is due before midnight on Monday, April 4.

In this assignment, we examine a Bayesian model of speech perception and implement an approximate inference algorithm. As discussed in lecture, a "speaker" produces a speech sound T (e.g., a vowel sound) intended for a "listener". Because of noise during tranmission, the listener doesn't hear T exactly; instead, he hears the corrupted physical stimulus S.

The model postulates that the listener does a type of active reconstruction. Instead of verbatim perception of S, the listener aims to reconstruct the intended utterance T. Concretely, his/her perceptual system estimates $P(T|S)$, and this reconstructed stimulus is what they actually "hear." Reconstructing the details of the intended production is a good idea for a listener, since these details can be important for understanding what was said due to co-articulation.

This Bayesian model aims to explain the "perceptual magnet effect" in speech perception. This effect describes a particular kind of warping due to categorical representations. The phenomenon is that the perceived sound is closer to the category center than the raw stimulus S, in a way likened to a "perceptual magnet." In Bayesian terms, we will model this as computing the reconstruction as the expected value of $P(T|S)$ (which is denoted $E[T|S]$). See the figure below for an example.

We strongly suggest you read the David MacKay chapter (in NYU Classes resources), especially the section on Metropolis-Hastings, before proceedings with this assignment:

- MacKay, D. (2003). Chapter 29: Monte Carlo Methods. In Information Theory, Inference, and Learning Algorithms.

The Bayesian model of the perceptual margnet effect was introduced in this paper:

- Feldman, N. H., & Griffiths, T. L. (2007). A rational account of the perceptual magnet effect. In Proceedings of the Annual Meeting of the Cognitive Science Society. (http://ling.umd.edu/~nhf/papers/PerceptualMagnet.pdf)

In [1]:
```
# Import the necessary packages
import numpy as np
import random
from scipy.stats import norm
from scipy.special import logsumexp
%matplotlib inline
import matplotlib
import matplotlib.pyplot as plt
```

Probabilistic model

Let's dive into the model details. First, we will need to set some key parameters.

In [2]:
```
# Key parameters we need for the probabilistic model
mu_c = 0 # category mean
sigma_c = 0.5 # category standard deviation
sigma2_c = sigma_c**2 # category variance
sigma_s = 0.4 # perceptual noise standard deviation
sigma2_s = sigma_s**2 # perceptual noise variance
X = np.linspace(-0.5,2,num=20) # points we are going to evaluate for war
ping
```

Prior

Lets start with the prior. This is the distribution of utterances a speaker says when producing a particular speech sound (e.g., a specific vowel). The prior distribution $P(T)$ on speaker productions T is modeled as a normal distribution

$$P(T) = N(\mu_c, \sigma_c^2).$$

This distribution is implemented in the `logprior_normal` function, which computes the log-probability. Although not necessary for this very simple case, it's important to ALWAYS compute with log-probabilities to prevent numerical underflow errors.

In [3]:
```
def logprior_normal(T):
    # Log-probability of speech production T
    return norm.logpdf(T,mu_c,sigma_c)
```

Likelihood

The production T is perturbed by noise to become the listener's perceived stimulus S. This noise process is also modeled as a normal distribution

$$P(S|T) = N(T, \sigma_S^2),$$

with the amount of noise governed by the standard deviation parameter σ_S. This distribution is implemented in `loglikelihood_normal`.

In [4]:
```
def loglikelihood_normal(S,T):
    # Log-probability of a stimulus S given production T
    return norm.logpdf(S,T,sigma_s)
```

Posterior mean, for model with normal prior and normal likelihood

In this Bayesian model, we assume that the goal of the listener is to optimally infer the intended production T given the perceived stimulus S. In other words, the listener is computing the posterior

$$P(T|S) = \frac{P(S|T)P(T)}{P(S)}.$$

As the prior and likelihood are normally distributed, we have a "conjugate prior", meaning the posterior takes the same distributional form as the prior. Thus $P(T|S)$ is also normally distributed. If you work out the math (a great exercise for those interested!), the posterior is

$$P(T|S) = N(\frac{\sigma_c^2 S + \sigma_S^2 \mu_c}{\sigma_c^2 + \sigma_S^2}, \frac{\sigma_c^2 \sigma_S^2}{\sigma_c^2 + \sigma_S^2}).$$

For the purposes of this assignment, we are only interested in the expected value (mean) of the posterior distribution, which is

$$E[T|S] = \frac{\sigma_c^2 S + \sigma_S^2 \mu_c}{\sigma_c^2 + \sigma_S^2},$$

corresponding to the "best guess" of the unobservable intended production T.

Notice that this is a weighted average between the actual stimulus S and the prior mean μ_c. The form of this average is intuitive. If the perceptual noise is high (high σ_S), the listener relies more on her prior expectations of about what the speech sound typically sounds like, giving μ_c a higher weight. If the prior expectation is highly variable (high σ_c), the listener relies more heavily on the perceived stimulus S. These are predictions the model makes, which have been empirically verified.

We provide the function `post_mean_normal_normal` for computing this "best guess" expected value of the posterior. This function is only valid when both, prior and likelihood, are each represented by a normal distribution.

In [5]:
```
def post_mean_normal_normal(S):
    # Posterior mean E[T|S] of sound production T given signal S, given
 normal P(T) and P(S|T)
    return (sigma2_c*S+sigma2_s*mu_c)/(sigma2_c+sigma2_s)
```

Visualizing the perceptual magnet effect

Let's see this Bayesian model in action. The `plot_warp` function visualizes how various raw stimulus values S warp to become perceived values $E[T|S]$. For its arguments, the parameter `stimuli_eval` is a numpy array of all of the values of S we want to evaluate. The function handle `f_posterior_mean` (see description below for a reminder about function handles) computes/estimates the posterior mean for a given stimulus S. The function handle `f_logprior` returns the log-probability of the prior for utterances T.

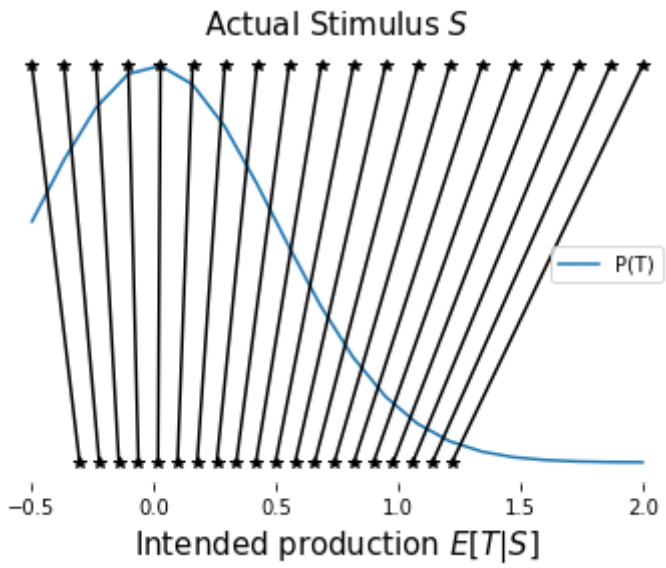
Let's run the `plot_warp` function for the normal-normal model that we have developed so far. The `post_mean_normal_normal` is the function handle for the posterior mean, and the function `logprior_normal` is the handle for the log prior.

This code produces a plot showing actual stimuli (S) at the top and perceived stimuli $E[T|S]$ at the bottom, overlaid with the category density $P(T)$ shown in blue. There is a clear "perceptual magnet effect" where perception warps the stimuli toward the category center.

Note: Function handles are pointers to a function which can be executed with the syntax `f_posterior_mean(S)` or `f_logprior(T)`. We need to pass handles, rather than pre-executed functions, so the script can control for itself when to execute the functions.

In [6]:
```
def plot_warp(f_posterior_mean,stimuli_eval,f_logprior,verbose=False):
    # Input
    #   f_posterior_mean : function handle f(S) that estimates posterior
mean E[T|S] for a raw stimulus S
    #      (only works for scalar S)
    #   stimuli_eval : [numpy array] of raw stimuli S we want to evalute
    #   f_logprior : function handle f(T) that evaluates log-prior log P
(T) for production T
    plt.figure()
    mypdf = np.exp(f_logprior(stimuli_eval))
    mx = np.max(mypdf)
    plt.plot(stimuli_eval,mypdf)
    for idx,x in enumerate(stimuli_eval):
        if verbose:
            print('  Estimating ' + str(idx+1) + ' of ' + str(len(stimul
i_eval)) + ' stimuli S')
        x_new = f_posterior_mean(x)
        plt.plot([x,x_new],[mx,0.],'k*-')
    plt.legend(['P(T)'])
    plt.tick_params(top=False, bottom=True, left=False, right=False, lab
elleft=False, labelbottom=True)
    for spine in plt.gca().spines.values():
        spine.set_visible(False)
    plt.title('Actual Stimulus $S$',size=15)
    plt.xlabel('Intended production $E[T|S]$',size=15)

print('Normal-normal model with exact inference')
plot_warp(post_mean_normal_normal,X,logprior_normal)
```

```
Normal-normal model with exact inference
```



Approximate inference with Metropolis-Hastings algorithm

So far, we have developed a simple normal-normal model, where the posterior mean can be computed in closed form, e.g., via `post_mean_normal_normal`. Usually, we are not so lucky, and a closed form solution is not available. In most cases, approximate inference algorithms are needed. Here, you will implement the very general and powerful Metropolis-Hasting algorithm for approximate inference using Markov Chain Monte Carlo (MCMC). See your lecture slides for the algorithm specification.

Metropolis-Hastings (MH) constructs a sequence of samples that converges to the posterior $P(T|S)$, if the sequence is run for long enough. At each step, a new value of T is proposed, and it is accepted or rejected based on its score using the MH "acceptance rule." Either way, the value of T is stored as a sample, and another proposal is made at the next step. A certain number of samples is thrown away at the beginning of the chain (burn in), and the remaining can be used to approximate the posterior $P(T|S)$. In this case, we want to estimate the posterior mean $E[T|S]$, so we average the samples with `np.mean(samples[nburn_in:])`.

Problem 1 (20 points)

Fill in the missing code below for a MH sampler for estimating E[T|S].

- There should be produces `nsamp` samples total, but with `nburn_in` samples thrown out from the beginning of the sequence.
- New proposals are made from a normal distribution centered at the current value of `T` with standard deviation `prop_width`.

More information on Metropolis-Hastings can be found in the David MacKay chapter on Monte Carlo methods in the Resources folder on NYU classes.

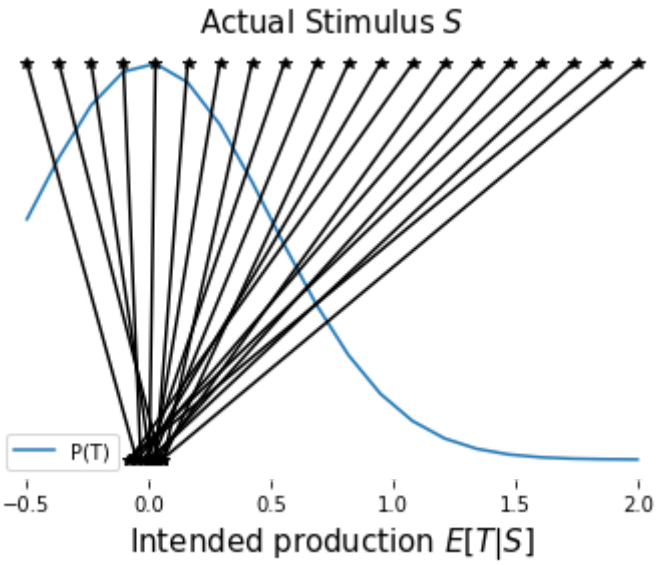
Hint: Computing the acceptance ratio a (see lecture slides) does not need to involve the normalizing constant $P(S)$ in the posterior,

$$P(T|S) = \frac{P(S|T)P(T)}{P(S)}.$$

This constant does not depend on the the variable T, the variable we are sampling, and it cancels out in the numerator and denominator when computing the ratio a. Thus it does not need to be included. This is critical since for many probabilistic models, we don't know the normalizing constant and thus can't use it in the acceptance ratio. This is one reason why MCMC is such a useful tool for probabilistic inference.

In [9]:
```
def estimate_metropolis_hastings(S,f_loglikelihood,f_logprior,nsamp=20(
,nburn_in=100,prop_width=0.25):
    #
    #  Draw a sequence of samples T_(nburn_in), T_2, ..., T_nsamp from t
he posterior distribution P(T|S)
    #
    # Input
    #  S : actual stimulus (scalar value only)
    #  f_loglikelihood : function handle f(S,T) to log-likelihood log P
(S|T)
    #  f_logprior : function handle f(t) to log-prior P(T)
    #  nsamp : how many samples to produce in MCMC chain
    #  nburn_in : how many samples at the beginning of the chain should
 we toss away
    #  prop_width : standard deviation of Gaussian proposal, centered at
current value of T
    #
    assert(isinstance(S, float))
    samples = []
    # TODO: Your code goes here

    mean = 0
    posterior = np.exp(f_loglikelihood(S, mean) + f_logprior(mean))

    while len(samples) < nsamp:
        prop = norm(mean, prop_width).rvs()
        proposalT = np.exp(f_loglikelihood(mean, prop)) * np.exp(f_logpr
ior(prop))

        accept_factor = proposalT / posterior
        if accept_factor >= 1:
            posterior = proposalT
            mean = prop
        elif random.random() < accept_factor:
            posterior = proposalT
            mean = prop
        samples.append(mean)

    return np.mean(samples[nburn_in:])

print('Normal-normal model with MCMC inference')
f_posterior_mean = lambda S : estimate_metropolis_hastings(S,loglikeliho
od_normal,logprior_normal)
plot_warp(f_posterior_mean,X,logprior_normal,verbose=True)
```

```
Normal-normal model with MCMC inference
  Estimating 1 of 20 stimuli S
  Estimating 2 of 20 stimuli S
  Estimating 3 of 20 stimuli S
  Estimating 4 of 20 stimuli S
  Estimating 5 of 20 stimuli S
  Estimating 6 of 20 stimuli S
  Estimating 7 of 20 stimuli S
  Estimating 8 of 20 stimuli S
  Estimating 9 of 20 stimuli S
  Estimating 10 of 20 stimuli S
  Estimating 11 of 20 stimuli S
  Estimating 12 of 20 stimuli S
  Estimating 13 of 20 stimuli S
  Estimating 14 of 20 stimuli S
  Estimating 15 of 20 stimuli S
  Estimating 16 of 20 stimuli S
  Estimating 17 of 20 stimuli S
  Estimating 18 of 20 stimuli S
  Estimating 19 of 20 stimuli S
  Estimating 20 of 20 stimuli S
```



Problem 2 (5 points)

Run your code from Problem 1 and examine the plot. Note that for each possible stimulus S, we run a different MCMC chain to estimate E[T|S].

- What is your reaction to the plot? How do the approximate estimates of E[T|S] using the sampler, compare to exact inference?
- What happens when you decrease the number of samples used in the MH algorithm?

YOUR RESPONSE GOES HERE

My reaction to the plot is that these graphs clearly show the perceptual magnet effect that occurs due to warping from categorical representations. This is shown on all plots where Stimulus S is mapped around 0 center. Compared to exact inference, MH algorithm's estimation of E[T|S] performs better even at lower sample size of 150.

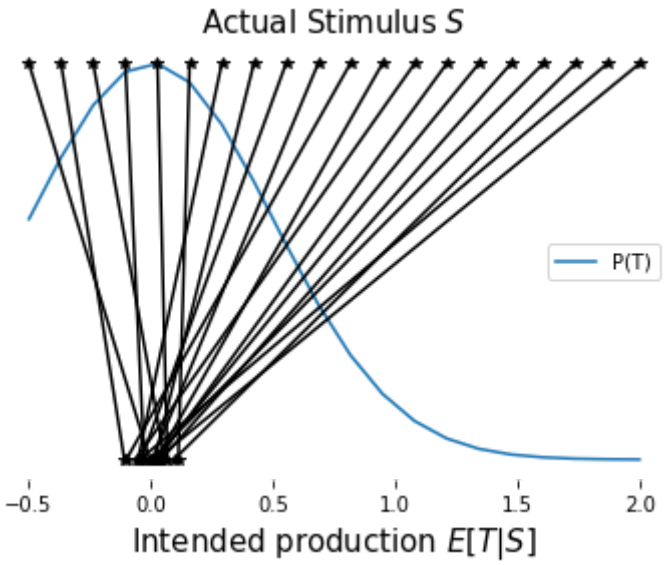
When I decrease the number of samples used in MH algorithm, perceived sound is not as concentrated on the category center as when larger samples are used. Due to the magnet effect, E[T|S] is mapped around 0.0 value (category center) more precisely at larger samples sizes. This is expected because more samples will create and demonstrate perceptual magnet effect much more clearly in a larger quantities.

Below graphs are created using different nsamp sizes at {1000, 500, 150}

In [18]:

```
# nsamp = 1000
def estimate_metropolis_hastings(S,f_loglikelihood,f_logprior,nsamp=1000
,nburn_in=100,prop_width=0.25):
    #
    #  Draw a sequence of samples T_(nburn_in), T_2, ..., T_nsamp from t
he posterior distribution P(T|S)
    #
    # Input
    #  S : actual stimulus (scalar value only)
    #  f_loglikelihood : function handle f(S,T) to log-likelihood log P
(S|T)
    #  f_logprior : function handle f(t) to log-prior P(T)
    #  nsamp : how many samples to produce in MCMC chain
    #  nburn_in : how many samples at the beginning of the chain should
 we toss away
    #  prop_width : standard deviation of Gaussian proposal, centered at
current value of T
    #
    assert(isinstance(S, float))
    samples = []
    # TODO: Your code goes here

    mean = 0
    posterior = np.exp(f_loglikelihood(S, mean) + f_logprior(mean))

    while len(samples) < nsamp:
        prop = norm(mean, prop_width).rvs()
        proposalT = np.exp(f_loglikelihood(mean, prop)) * np.exp(f_logpr
ior(prop))

        accept_factor = proposalT / posterior
        if accept_factor >= 1:
            posterior = proposalT
            mean = prop
        elif random.random() < accept_factor:
            posterior = proposalT
            mean = prop
        samples.append(mean)

    return np.mean(samples[nburn_in:])

print('Normal-normal model with MCMC inference')
f_posterior_mean = lambda S : estimate_metropolis_hastings(S,loglikeliho
od_normal,logprior_normal)
plot_warp(f_posterior_mean,X,logprior_normal,verbose=True)
```

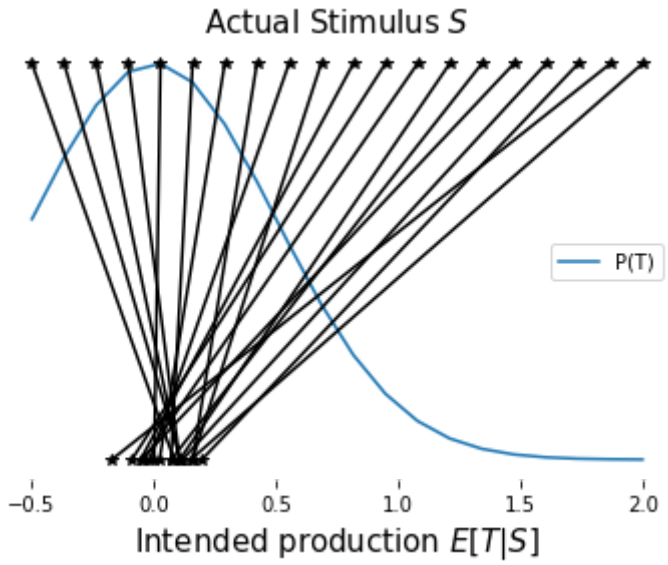
```
Normal-normal model with MCMC inference
  Estimating 1 of 20 stimuli S
  Estimating 2 of 20 stimuli S
  Estimating 3 of 20 stimuli S
  Estimating 4 of 20 stimuli S
  Estimating 5 of 20 stimuli S
  Estimating 6 of 20 stimuli S
  Estimating 7 of 20 stimuli S
  Estimating 8 of 20 stimuli S
  Estimating 9 of 20 stimuli S
  Estimating 10 of 20 stimuli S
  Estimating 11 of 20 stimuli S
  Estimating 12 of 20 stimuli S
  Estimating 13 of 20 stimuli S
  Estimating 14 of 20 stimuli S
  Estimating 15 of 20 stimuli S
  Estimating 16 of 20 stimuli S
  Estimating 17 of 20 stimuli S
  Estimating 18 of 20 stimuli S
  Estimating 19 of 20 stimuli S
  Estimating 20 of 20 stimuli S
```



In [16]:

```
# nsamp = 500
def estimate_metropolis_hastings(S,f_loglikelihood,f_logprior,nsamp=500,
nburn_in=100,prop_width=0.25):
    #
    #  Draw a sequence of samples T_(nburn_in), T_2, ..., T_nsamp from t
he posterior distribution P(T|S)
    #
    # Input
    #  S : actual stimulus (scalar value only)
    #  f_loglikelihood : function handle f(S,T) to log-likelihood log P
(S|T)
    #  f_logprior : function handle f(t) to log-prior P(T)
    #  nsamp : how many samples to produce in MCMC chain
    #  nburn_in : how many samples at the beginning of the chain should
 we toss away
    #  prop_width : standard deviation of Gaussian proposal, centered at
current value of T
    #
    assert(isinstance(S, float))
    samples = []
    # TODO: Your code goes here

    mean = 0
    posterior = np.exp(f_loglikelihood(S, mean) + f_logprior(mean))

    while len(samples) < nsamp:
        prop = norm(mean, prop_width).rvs()
        proposalT = np.exp(f_loglikelihood(mean, prop)) * np.exp(f_logpr
ior(prop))

        accept_factor = proposalT / posterior
        if accept_factor >= 1:
            posterior = proposalT
            mean = prop
        elif random.random() < accept_factor:
            posterior = proposalT
            mean = prop
        samples.append(mean)

    return np.mean(samples[nburn_in:])

print('Normal-normal model with MCMC inference')
f_posterior_mean = lambda S : estimate_metropolis_hastings(S,loglikeliho
od_normal,logprior_normal)
plot_warp(f_posterior_mean,X,logprior_normal,verbose=True)
```

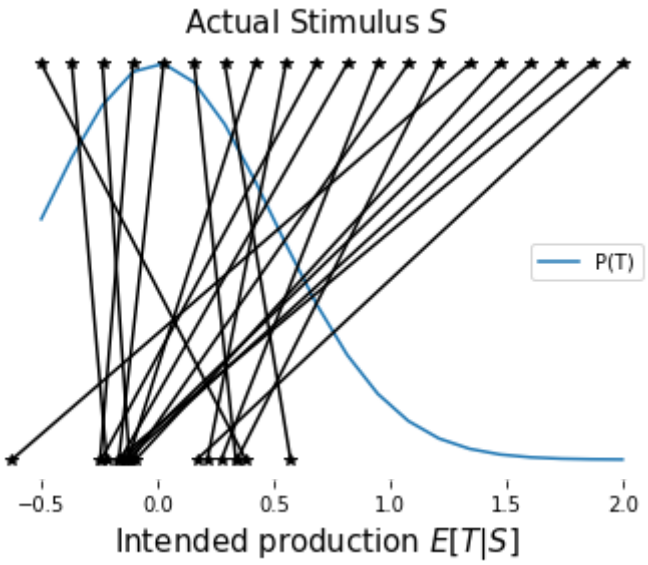
```
Normal-normal model with MCMC inference
  Estimating 1 of 20 stimuli S
  Estimating 2 of 20 stimuli S
  Estimating 3 of 20 stimuli S
  Estimating 4 of 20 stimuli S
  Estimating 5 of 20 stimuli S
  Estimating 6 of 20 stimuli S
  Estimating 7 of 20 stimuli S
  Estimating 8 of 20 stimuli S
  Estimating 9 of 20 stimuli S
  Estimating 10 of 20 stimuli S
  Estimating 11 of 20 stimuli S
  Estimating 12 of 20 stimuli S
  Estimating 13 of 20 stimuli S
  Estimating 14 of 20 stimuli S
  Estimating 15 of 20 stimuli S
  Estimating 16 of 20 stimuli S
  Estimating 17 of 20 stimuli S
  Estimating 18 of 20 stimuli S
  Estimating 19 of 20 stimuli S
  Estimating 20 of 20 stimuli S
```



In [17]:

```
# nsamp = 150
def estimate_metropolis_hastings(S,f_loglikelihood,f_logprior,nsamp=150,
nburn_in=100,prop_width=0.25):
    #
    #  Draw a sequence of samples T_(nburn_in), T_2, ..., T_nsamp from t
he posterior distribution P(T|S)
    #
    # Input
    #  S : actual stimulus (scalar value only)
    #  f_loglikelihood : function handle f(S,T) to log-likelihood log P
(S|T)
    #  f_logprior : function handle f(t) to log-prior P(T)
    #  nsamp : how many samples to produce in MCMC chain
    #  nburn_in : how many samples at the beginning of the chain should
 we toss away
    #  prop_width : standard deviation of Gaussian proposal, centered at
current value of T
    #
    assert(isinstance(S, float))
    samples = []
    # TODO: Your code goes here

    mean = 0
    posterior = np.exp(f_loglikelihood(S, mean) + f_logprior(mean))

    while len(samples) < nsamp:
        prop = norm(mean, prop_width).rvs()
        proposalT = np.exp(f_loglikelihood(mean, prop)) * np.exp(f_logpr
ior(prop))

        accept_factor = proposalT / posterior
        if accept_factor >= 1:
            posterior = proposalT
            mean = prop
        elif random.random() < accept_factor:
            posterior = proposalT
            mean = prop
        samples.append(mean)

    return np.mean(samples[nburn_in:])

print('Normal-normal model with MCMC inference')
f_posterior_mean = lambda S : estimate_metropolis_hastings(S,loglikeliho
od_normal,logprior_normal)
plot_warp(f_posterior_mean,X,logprior_normal,verbose=True)
```

```
Normal-normal model with MCMC inference
  Estimating 1 of 20 stimuli S
  Estimating 2 of 20 stimuli S
  Estimating 3 of 20 stimuli S
  Estimating 4 of 20 stimuli S
  Estimating 5 of 20 stimuli S
  Estimating 6 of 20 stimuli S
  Estimating 7 of 20 stimuli S
  Estimating 8 of 20 stimuli S
  Estimating 9 of 20 stimuli S
  Estimating 10 of 20 stimuli S
  Estimating 11 of 20 stimuli S
  Estimating 12 of 20 stimuli S
  Estimating 13 of 20 stimuli S
  Estimating 14 of 20 stimuli S
  Estimating 15 of 20 stimuli S
  Estimating 16 of 20 stimuli S
  Estimating 17 of 20 stimuli S
  Estimating 18 of 20 stimuli S
  Estimating 19 of 20 stimuli S
  Estimating 20 of 20 stimuli S
```



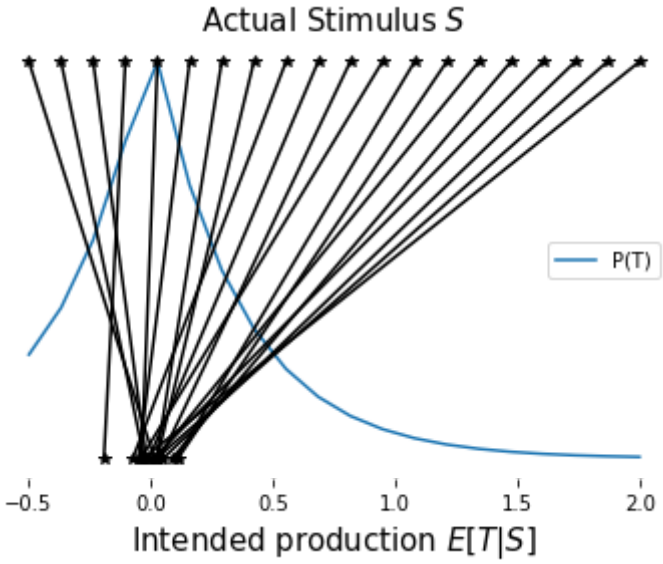
Non-conjugate Bayesian model of speech perception

To demonstrate the power and generality of the Metropolis-Hastings algorithm, let's change the probabilistic model. Rather than using a normal distribution $P(T)$ over speaker utterances, let's assume we have a Laplace distribution (https://en.wikipedia.org/wiki/Laplace_distribution) instead. It is unimodal like a normal distribution, but with fatter tails.

Use the code below to make a new plot. This time, we use the Laplace prior `logprior_laplace` over speaker utterances instead of the normal prior over utterances.

In [19]:
```
def logprior_laplace(T):
    # Alternative prior distribution
    #  log P(T|c) ~ Laplace(mu_c,b)
    b = sigma_c/np.sqrt(2)
    return (-np.abs(T-mu_c)/b) - np.log(2*b)

print('Laplace-normal model with MCMC inference')
f_posterior_mean = lambda S : estimate_metropolis_hastings(S,loglikeliho
od_normal,logprior_laplace)
plot_warp(f_posterior_mean,X,logprior_laplace,verbose=True)
```

```
Laplace-normal model with MCMC inference
  Estimating 1 of 20 stimuli S
  Estimating 2 of 20 stimuli S
  Estimating 3 of 20 stimuli S
  Estimating 4 of 20 stimuli S
  Estimating 5 of 20 stimuli S
  Estimating 6 of 20 stimuli S
  Estimating 7 of 20 stimuli S
  Estimating 8 of 20 stimuli S
  Estimating 9 of 20 stimuli S
  Estimating 10 of 20 stimuli S
  Estimating 11 of 20 stimuli S
  Estimating 12 of 20 stimuli S
  Estimating 13 of 20 stimuli S
  Estimating 14 of 20 stimuli S
  Estimating 15 of 20 stimuli S
  Estimating 16 of 20 stimuli S
  Estimating 17 of 20 stimuli S
  Estimating 18 of 20 stimuli S
  Estimating 19 of 20 stimuli S
  Estimating 20 of 20 stimuli S
```



Problem 3 (5 points)

- What effect did replacing the prior have on the model?
- Is there a perceptual magnet effect with the Laplace prior? Does the Bayesian explanation of the phenomenon depend on having a normal prior?

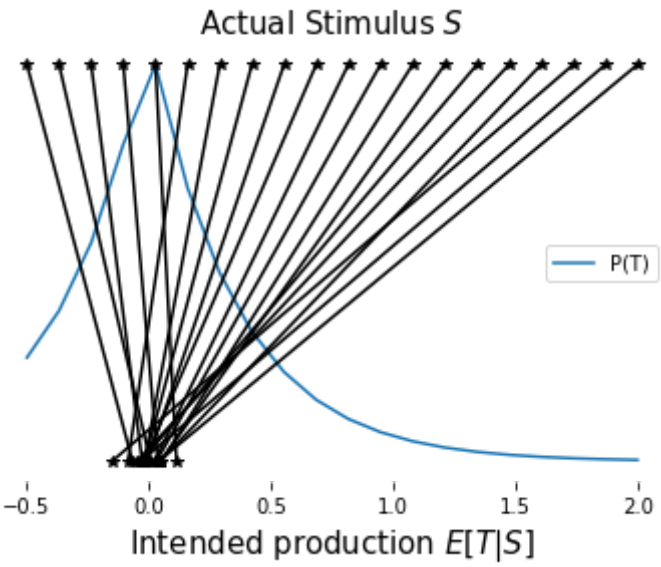
YOUR RESPONSE HERE

Replacing the prior using Laplace from normal had a small improvement, but it is difficult to say it made a siginificant difference. The accuracy was very similar in both Laplace and normal. This is because at larger sample sizes, both algorithms work very well, and it is only at the 150 sample size that we perceive less outliers away from the 0 center.

Perceptual magenet effect was also present with clear visualization in the Laplace prior as well. This means that Bayesian explanation of the phenomenon doesn't depend on having a normal prior, and also the accuracy stays similar for the most part.

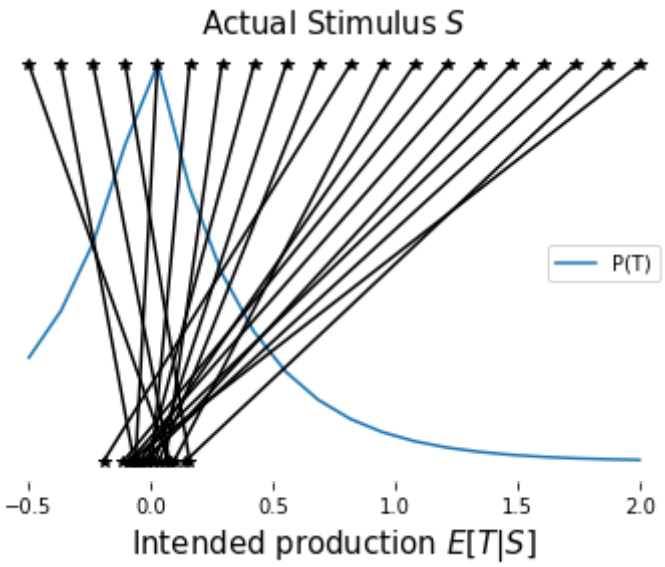
In [20]:
```
print('Laplace-normal model with MCMC inference @ nsamp=2000')
f_posterior_mean = lambda S : estimate_metropolis_hastings(S,loglikeliho
od_normal,logprior_laplace, 2000)
plot_warp(f_posterior_mean,X,logprior_laplace,verbose=True)
```

```
Laplace-normal model with MCMC inference @ nsamp=2000
  Estimating 1 of 20 stimuli S
  Estimating 2 of 20 stimuli S
  Estimating 3 of 20 stimuli S
  Estimating 4 of 20 stimuli S
  Estimating 5 of 20 stimuli S
  Estimating 6 of 20 stimuli S
  Estimating 7 of 20 stimuli S
  Estimating 8 of 20 stimuli S
  Estimating 9 of 20 stimuli S
  Estimating 10 of 20 stimuli S
  Estimating 11 of 20 stimuli S
  Estimating 12 of 20 stimuli S
  Estimating 13 of 20 stimuli S
  Estimating 14 of 20 stimuli S
  Estimating 15 of 20 stimuli S
  Estimating 16 of 20 stimuli S
  Estimating 17 of 20 stimuli S
  Estimating 18 of 20 stimuli S
  Estimating 19 of 20 stimuli S
  Estimating 20 of 20 stimuli S
```



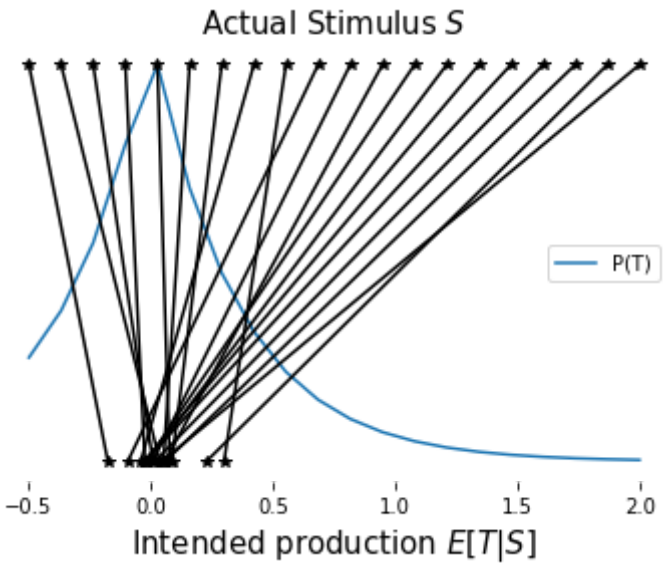
In [21]:
```
print('Laplace-normal model with MCMC inference @ nsamp=1000')
f_posterior_mean = lambda S : estimate_metropolis_hastings(S,loglikeliho
od_normal,logprior_laplace, 1000)
plot_warp(f_posterior_mean,X,logprior_laplace,verbose=True)
```

```
Laplace-normal model with MCMC inference @ nsamp=1000
  Estimating 1 of 20 stimuli S
  Estimating 2 of 20 stimuli S
  Estimating 3 of 20 stimuli S
  Estimating 4 of 20 stimuli S
  Estimating 5 of 20 stimuli S
  Estimating 6 of 20 stimuli S
  Estimating 7 of 20 stimuli S
  Estimating 8 of 20 stimuli S
  Estimating 9 of 20 stimuli S
  Estimating 10 of 20 stimuli S
  Estimating 11 of 20 stimuli S
  Estimating 12 of 20 stimuli S
  Estimating 13 of 20 stimuli S
  Estimating 14 of 20 stimuli S
  Estimating 15 of 20 stimuli S
  Estimating 16 of 20 stimuli S
  Estimating 17 of 20 stimuli S
  Estimating 18 of 20 stimuli S
  Estimating 19 of 20 stimuli S
  Estimating 20 of 20 stimuli S
```



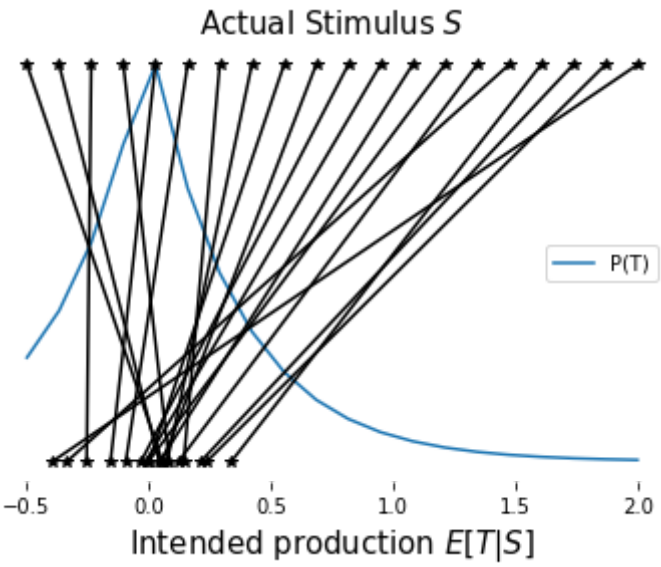
In [22]:
```
print('Laplace-normal model with MCMC inference @ nsamp=500')
f_posterior_mean = lambda S : estimate_metropolis_hastings(S,loglikeliho
od_normal,logprior_laplace, 500)
plot_warp(f_posterior_mean,X,logprior_laplace,verbose=True)
```

```
Laplace-normal model with MCMC inference @ nsamp=500
  Estimating 1 of 20 stimuli S
  Estimating 2 of 20 stimuli S
  Estimating 3 of 20 stimuli S
  Estimating 4 of 20 stimuli S
  Estimating 5 of 20 stimuli S
  Estimating 6 of 20 stimuli S
  Estimating 7 of 20 stimuli S
  Estimating 8 of 20 stimuli S
  Estimating 9 of 20 stimuli S
  Estimating 10 of 20 stimuli S
  Estimating 11 of 20 stimuli S
  Estimating 12 of 20 stimuli S
  Estimating 13 of 20 stimuli S
  Estimating 14 of 20 stimuli S
  Estimating 15 of 20 stimuli S
  Estimating 16 of 20 stimuli S
  Estimating 17 of 20 stimuli S
  Estimating 18 of 20 stimuli S
  Estimating 19 of 20 stimuli S
  Estimating 20 of 20 stimuli S
```



In [23]:
```
print('Laplace-normal model with MCMC inference @ nsamp=150')
f_posterior_mean = lambda S : estimate_metropolis_hastings(S,loglikeliho
od_normal,logprior_laplace, 150)
plot_warp(f_posterior_mean,X,logprior_laplace,verbose=True)
```

```
Laplace-normal model with MCMC inference @ nsamp=150
  Estimating 1 of 20 stimuli S
  Estimating 2 of 20 stimuli S
  Estimating 3 of 20 stimuli S
  Estimating 4 of 20 stimuli S
  Estimating 5 of 20 stimuli S
  Estimating 6 of 20 stimuli S
  Estimating 7 of 20 stimuli S
  Estimating 8 of 20 stimuli S
  Estimating 9 of 20 stimuli S
  Estimating 10 of 20 stimuli S
  Estimating 11 of 20 stimuli S
  Estimating 12 of 20 stimuli S
  Estimating 13 of 20 stimuli S
  Estimating 14 of 20 stimuli S
  Estimating 15 of 20 stimuli S
  Estimating 16 of 20 stimuli S
  Estimating 17 of 20 stimuli S
  Estimating 18 of 20 stimuli S
  Estimating 19 of 20 stimuli S
  Estimating 20 of 20 stimuli S
```



In [ ]: