# Demonstration of Linear Bayesian Regression

Code adapted from: https://zjost.github.io/bayesian-linear-regression/

In [1]:

```
import numpy as np
from numpy.random import normal, uniform
from scipy.stats import multivariate_normal as mv_norm
import matplotlib.pyplot as plt
%matplotlib inline

def real_function(a_0, a_1, noise_sigma, x):
    """
    Evaluates the real function
    """
    N = len(x)
    if noise_sigma==0:
        # Recovers the true function
        return a_0 + a_1*x
    else:
        return a_0 + a_1*x + normal(0, noise_sigma, N)

class LinearBayes(object):
    """
    A class that holds parameter prior/posterior and handles
    the hyper-parameter updates with new data

    Note:  variables starting with "v_" indicate Nx1 dimensional
        column vectors, those starting with "m_" indicate
        matrices, and those starting with "a_" indicate
        1xN dimensional arrays.

    Args:
        a_m0 (np.array): prior mean vector of size 1xM
        m_S0 (np.ndarray): prior covariance matrix of size MxM
        beta (float): known real-data noise precision

    """
    def __init__(self, a_m0, m_S0, beta):
        self.prior = mv_norm(mean=a_m0, cov=m_S0)
```

```
        self.v_m0 = a_m0.reshape(a_m0.shape + (1,)) #reshape to column
vector
        self.m_S0 = m_S0
        self.beta = beta

        self.v_mN = self.v_m0
        self.m_SN = self.m_S0
        self.posterior = self.prior

    def get_params(self):
        return self.v_mN

    def get_phi(self, a_x):
        """
        Returns the design matrix of size (NxM) for a feature vector v_x.
        In this case, this function merely adds the phi_0 dummy basis
        that's equal to 1 for all elements.

        Args:
            a_x (np.array): input features of size 1xN
        """
        m_phi = np.ones((len(a_x), 2))
        m_phi[:, 1] = a_x
        return m_phi

    def set_posterior(self, a_x, a_t):
        """
        Updates mN and SN given vectors of x-values and t-values
        """
        # Need to convert v_t from an array into a column vector
        # to correctly compute matrix multiplication
        v_t = a_t.reshape(a_t.shape + (1,))

        m_phi = self.get_phi(a_x)

        self.m_SN = np.linalg.inv(np.linalg.inv(self.m_S0) +
self.beta*m_phi.T.dot(m_phi))

        self.v_mN = self.m_SN.dot(np.linalg.inv(self.m_S0).dot(self.v_m0) +
\
```

```
                                    self.beta*m_phi.T.dot(v_t))

        self.posterior = mv_norm(mean=self.v_mN.flatten(), cov=self.m_SN)


    def prediction_limit(self, a_x, stdevs):
        """
        Calculates the limit that's "stdevs" standard deviations
        away from the mean at a given value of x.

        Args:
            a_x (np.array): x-axis values of size 1xN
            stdevs (float): Number of standard deviations away from
                the mean to calculate the prediction limit

        Returns:
            np.array: the prediction limit "stdevs" standard deviations
                away from the mean corresponding to x-values in "v_x"

        """
        N = len(a_x)
        m_x = self.get_phi(a_x).T.reshape((2, 1, N))

        predictions = []
        for idx in range(N):
            x = m_x[:,:,idx]
            sig_sq_x = 1/self.beta + x.T.dot(self.m_SN.dot(x))
            mean_x = self.v_mN.T.dot(x)
            predictions.append((mean_x+stdevs*np.sqrt(sig_sq_x)).flatten())
        return np.concatenate(predictions)

    def generate_data(self, a_x):
        N = len(a_x)
        m_x = self.get_phi(a_x).T.reshape((2, 1, N))

        predictions = []
        for idx in range(N):
            x = m_x[:,:,idx]
            sig_sq_x = 1/self.beta + x.T.dot(self.m_SN.dot(x))
            mean_x = self.v_mN.T.dot(x)
```

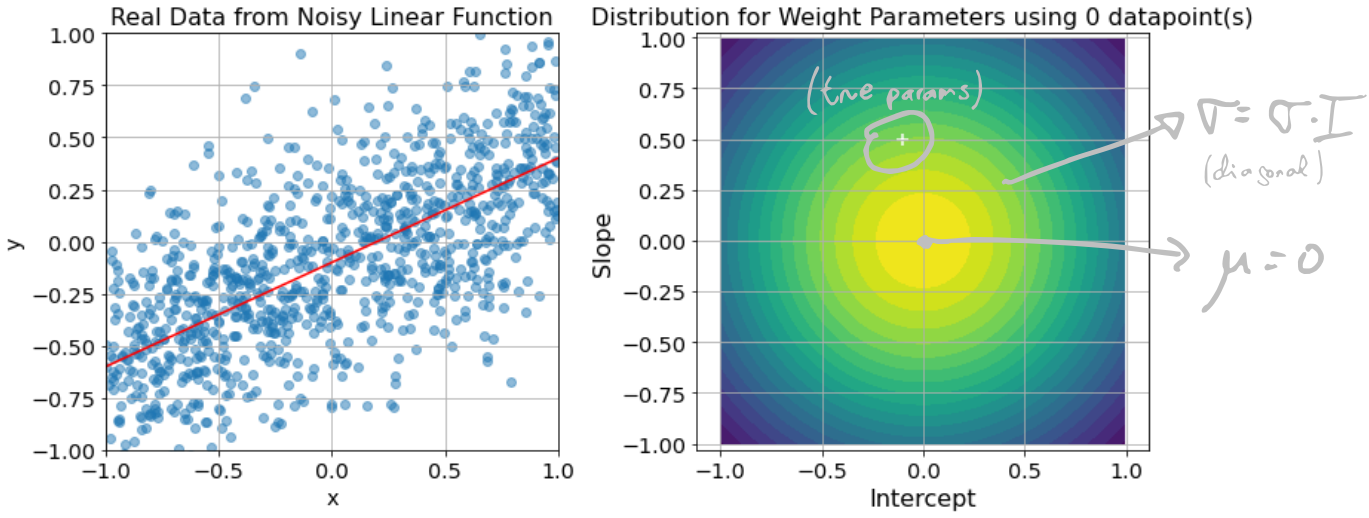
```
            predictions.append(normal(mean_x.flatten(), np.sqrt(sig_sq_x)))
        return np.array(predictions)

    def make_contour(self, a_x, a_y, real_parms=[], N=0):
        """
        A helper function to generate contour plots of our probability
distribution
        """
        pos = np.empty(a_x.shape + (2,))
        pos[:, :, 0] = a_x
        pos[:, :, 1] = a_y

        plt.contourf(a_x, a_y, self.posterior.pdf(pos), 20)
        plt.xlabel('Intercept', fontsize=16)
        plt.ylabel('Slope', fontsize=16)

        if real_parms:
            plt.scatter(real_parms[0], real_parms[1], marker='+',
c='white', s=60)

        _ = plt.title('Distribution for Weight Parameters using %d
datapoint(s)' % N, fontsize=16)


    def make_scatter(self, a_x, a_t, real_parms, samples=None,
stdevs=None):
        """
        A helper function to plot noisey data, the true function,
        and optionally a set of lines specified by the nested array of
        weights of size NxM where N is number of lines, M is 2 for
        this simple model
        """
        plt.scatter(a_x, a_t, alpha=0.5)
        plt.xlabel('x')
        plt.ylabel('y')

        plt.plot([-1, 1], real_function(real_parms[0], real_parms[1], 0,
np.array([-1., 1.])), 'r')

        _ = plt.title('Real Data from Noisy Linear Function', fontsize=16)
```

```
        if samples:
            weights = self.posterior.rvs(samples)
            for weight in weights:
                plt.plot([-1, 1], real_function(weight[0], weight[1], 0,
np.array([-1., 1.])), 'black')
                _ = plt.title('Lines Sampled from Posterior Distribution')

        if stdevs:
            a_xrange = np.linspace(-1, 1, 100)
            y_upper = self.prediction_limit(a_xrange, stdevs)
            y_lower = self.prediction_limit(a_xrange, -stdevs)
            plt.plot(a_xrange, y_upper, '+', c='green', linewidth=4.0)
            plt.plot(a_xrange, y_lower, '+', c='green', linewidth=4.0)
            _ = plt.title('Lines Sampled from Posterior Distribution')
```

In [3]:

```
# The following makes nice looking plots
params = {'legend.fontsize': 'x-large','figure.figsize': (12, 5),
          'axes.labelsize': 'x-large','axes.titlesize':'x-large',
          'xtick.labelsize':'x-large','ytick.labelsize':'x-large'}
plt.rcParams.update(params)

# CREATE DATASET
# Define true parameters
a_0 = -0.1
a_1 = 0.5
sigma2_epsilon = 0.1

# Generate input features
#np.random.seed(10) # Set the seed so we can get reproducible results
x_real = uniform(-1, 1, 1000)
# Evaluate the real function at these points
y_real = real_function(a_0, a_1, np.sqrt(sigma2_epsilon), x_real)

# define parameters of the parameter distribution
sigma2_w = 1
mean_w = np.array([0., 0.])
Sigma_w = np.identity(2)*sigma2_w
```

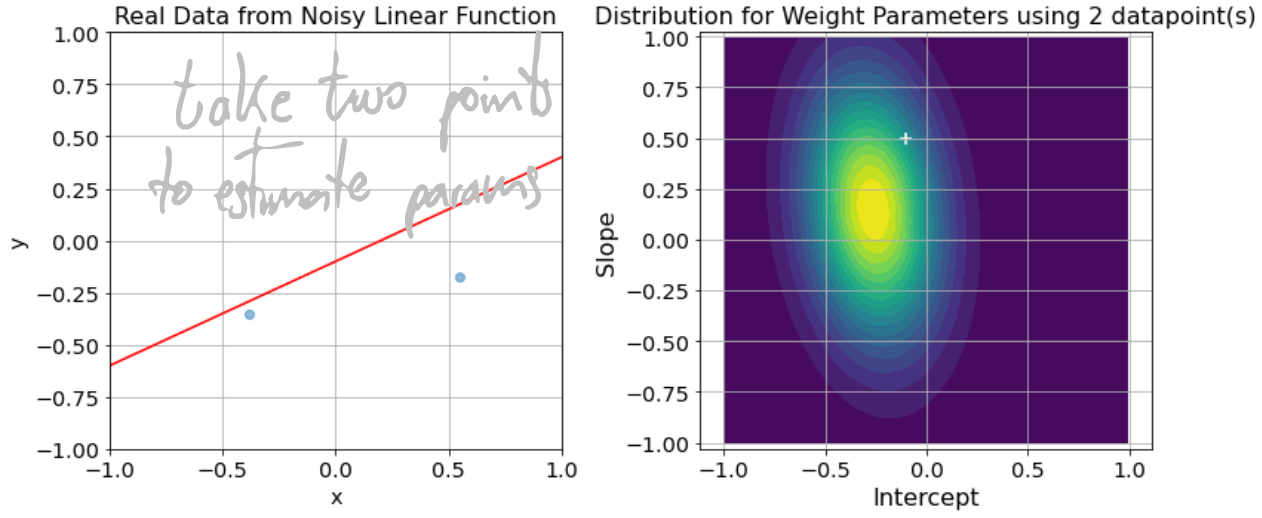
```
# Initialize the parameter model distribution
linbayes = LinearBayes(mean_w, Sigma_w, 1/sigma2_epsilon)

# Plot of our data against the true line
fig, ax1 = plt.subplots()
ax1 = plt.subplot(1, 2, 1)
linbayes.make_scatter(x_real, y_real, real_parms = [a_0, a_1])
ax1.set_xlim(left=-1,right=1)
ax1.set_ylim(top=1.0,bottom=-1.0)
ax1.grid(True)

# Plot the distribution of w, and the true w
ax2 = plt.subplot(1, 2, 2)
x, y = np.mgrid[-1:1:.01, -1:1:.01]
linbayes.make_contour(x, y, real_parms=[a_0, a_1], N=0)
ax2.grid(True)
ax2.set_xlim(left=-1,right=1)
ax2.axis('equal')
fig.tight_layout()

# save figures
#plt.savefig("bayesian_linear_regression_00.png")
#files.download("bayesian_linear_regression_00.png")
```




In [6]:

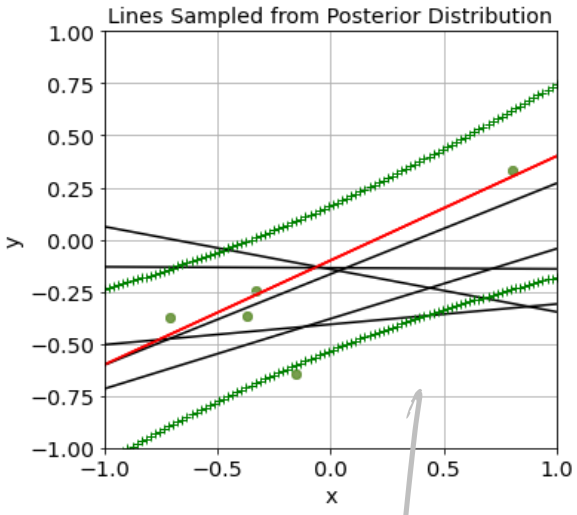
```
# How many points do we want to use to estimate parameters?
N=2

# initialize model
linbayes = LinearBayes(mean_w, Sigma_w, 1/sigma2_epsilon)
```

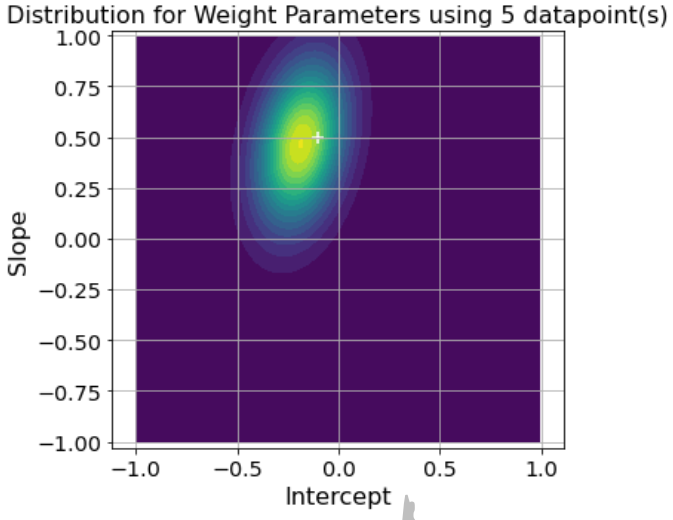
```
# Sample N indices of points from dataset
idx = np.random.choice(range(len(x_real)),N)

# Plot of our data against the true line
fig, ax1 = plt.subplots()
ax1 = plt.subplot(1, 2, 1)
linbayes.make_scatter(x_real[idx], y_real[idx], real_parms=[a_0, a_1])
ax1.set_xlim(left=-1,right=1)
ax1.set_ylim(top=1.0,bottom=-1.0)
ax1.grid(True)

# compute the posterior
linbayes.set_posterior(x_real[idx], y_real[idx])

# Plot the distribution of w, and the true w
ax2 = plt.subplot(1, 2, 2)
x, y = np.mgrid[-1:1:.01, -1:1:.01]
linbayes.make_contour(x, y, real_parms=[a_0, a_1], N=N)
ax2.grid(True)
ax2.set_xlim(left=-1,right=1)
ax2.set_ylim(bottom=-1,top=1)
ax2.axis('equal')
fig.tight_layout()

# save figures
#plt.savefig("bayesian_linear_regression_"+np.str(N)+".png")
#files.download("bayesian_linear_regression_"+np.str(N)+".png")
```



In [13]:

```
# How many points do we want to use to estimate parameters?
N=5

# initialize model
linbayes = LinearBayes(mean_w, Sigma_w, 1/sigma2_epsilon)

# Sample N indices of points from dataset
idx = np.random.choice(range(len(x_real)),N)

# compute the posterior
linbayes.set_posterior(x_real[idx], y_real[idx])

# Plot of our data against the true line
fig, ax1 = plt.subplots()
ax1 = plt.subplot(1, 2, 1)
linbayes.make_scatter(x_real[idx], y_real[idx], real_parms=[a_0, a_1])
linbayes.make_scatter(x_real[idx], y_real[idx], real_parms=[a_0, a_1],
samples=5)
linbayes.make_scatter(x_real[idx], y_real[idx], real_parms=[a_0, a_1],
stdevs=1)
ax1.set_xlim(left=-1,right=1)
ax1.set_ylim(top=1.0,bottom=-1.0)
ax1.grid(True)

# Plot the distribution of w, and the true w
ax2 = plt.subplot(1, 2, 2)
x, y = np.mgrid[-1:1:.01, -1:1:.01]
linbayes.make_contour(x, y, real_parms=[a_0, a_1], N=N)
ax2.grid(True)
ax2.set_xlim(left=-1,right=1)
ax2.set_ylim(bottom=-1,top=1)
ax2.axis('equal')
fig.tight_layout()

# save figures
#plt.savefig("bayesian_linear_regression_"+np.str(N)+".png")
#files.download("bayesian_linear_regression_"+np.str(N)+".png")
```




In [ ]:

More observations will lead to parameters predictions with lower uncertainty and higher accuracy.