Bayesian Hierarchical Modeling and HMC to Analyzing Ship Speed Data

Introduction

Bayesian hierarchical modeling is a statistical model written in multiple levels form that estimates the parameters of posterior distribution using the Bayesian method. In a short word, a hierarchical model is a case of multilevel model where parameters are nested within another. In real-world, hierarchical models are used when information is available on several different levels of observations. The methods of hierarchical analyzing play a significant role in the development of computational strategies. As for HMC, it's an instance of Markov chain Monte Carlo (MCMC) method for acquiring a sequence of random samples which converge to being distributed according to a target probability density for which direct sampling is pretty hard. HMC reduces the correlation between successive sampled states by proposing moves to distant states which maintain a high probability of acceptance due to the approximate energy conserving properties of the simulated Hamiltonian dynamic when using a symplectic integrator. In this project, we will compare the hierarchical modeling with non-hierarchical modeling and use stan's HMC simulate function to analyze the positions' influence on the speed of a ship.

Data description:

We use the data of the ship track and focus on the relation between speed and position of this ship. It has about 127078 rows and 15 features, each feature represents one column in the whole dataset. Cause we only focus on the positions effect to speed in this project, we only need to analyze the features 'LON' and 'LAT', which denote the longitude and latitude, respectively. First, let us upload csv file to python dataframe and print the first ten rows:

```
In [1]: import pandas as pd
        def read_data(file_name='ship.csv'):
            data = pd.read_csv('./code/{}'.format(file_name))
            data.dropna(how='any')
            return data

        data = read_data()
        print('The length of ship.csv data is {}'.format(len(data)))
        data.head(6)
```

```
The length of ship.csv data is 127078
```

Out[1]:

	Var1	MMSI	SHIP_ID	STATUS	SPEED_KNOTSx10	LON	LAT	COURSE	HEADING	TII
0	727	276829000	4691590	5	0	24.770250	59.444149	15	206.0	
1	736	276829000	4691590	5	0	24.770000	59.443329	203	NaN	
2	1628	276829000	4691590	0	226	24.696449	59.561600	159	160.0	
3	1640	276829000	4691590	0	225	24.715771	59.663120	210	210.0	
4	1641	276829000	4691590	0	224	24.744619	59.688480	209	210.0	
5	1651	276829000	4691590	0	227	24.794331	59.772049	193	194.0	
6	2601	276829000	4691590	0	225	24.716370	59.535412	158	160.0	
7	2617	276829000	4691590	0	224	24.729851	59.675549	210	210.0	
8	2623	276829000	4691590	0	227	24.773279	59.728069	193	194.0	
9	3251	276829000	4691590	0	227	24.786650	59.756149	194	194.0	

Then, we remove the unralted features and just concentrate on the SPEED, LON, LAT.

In [2]:
```
data = data[['SPEED_KNOTSx10', 'LON', 'LAT']]
data = data.rename(columns={'SPEED_KNOTSx10': 'SPEED'})
data.head(5)
```

Out[2]:

	SPEED	LON	LAT
0	0	24.770250	59.444149
1	0	24.770000	59.443329
2	226	24.696449	59.561600
3	225	24.715771	59.663120
4	224	24.744619	59.688480

Furthermore, we need to delete the rows that SPEED is equal to zero. Because when SPEED is zero that means ship are in on rest, we do not to regrad it as the information of sailing. Also, we need to divide the SPEED by 10 cause the current value of SPEED is the 10 times of speed knots.

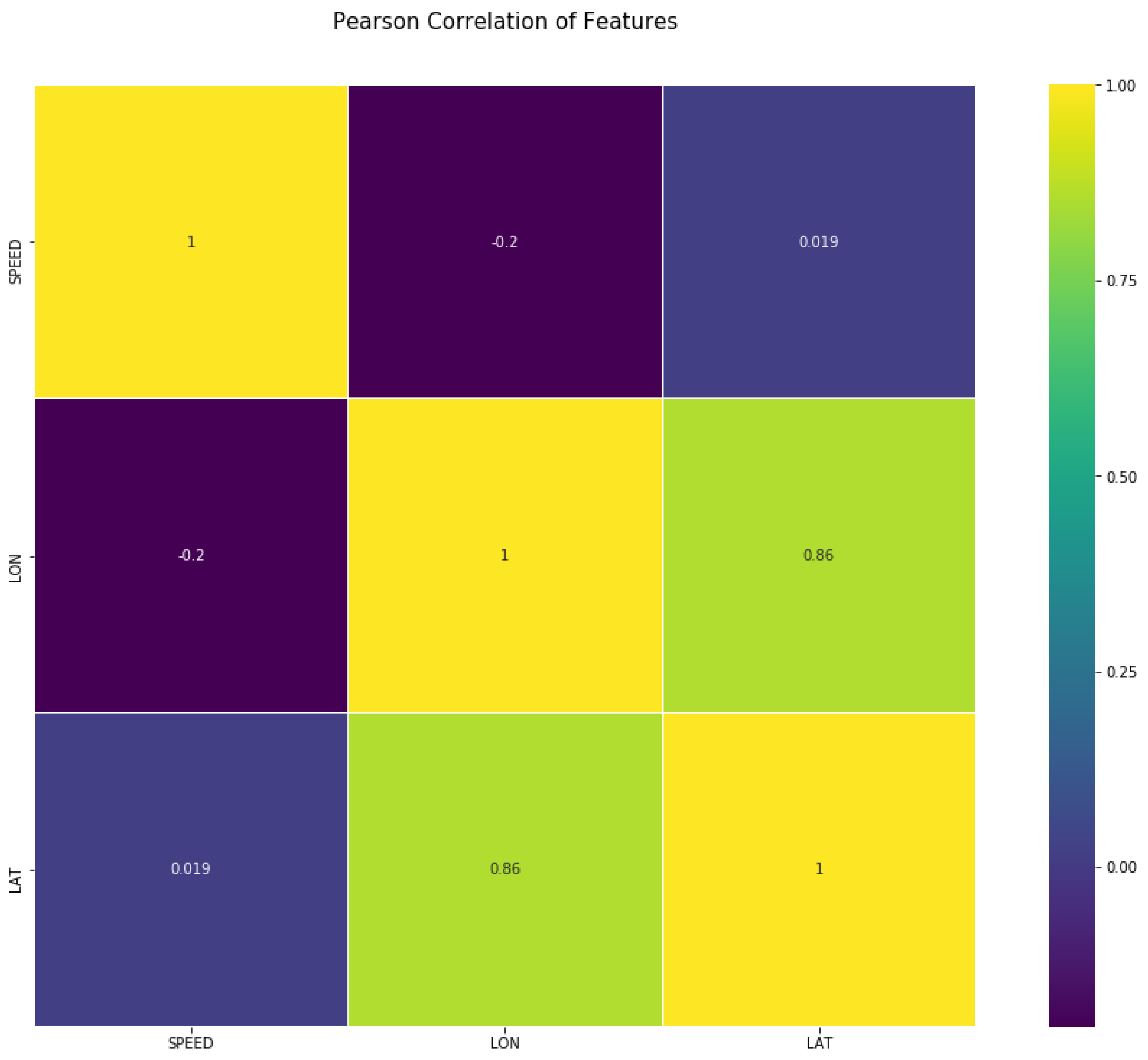
In [3]:
```
data = data[data.SPEED != 0]
data.SPEED /= 10

data.head(5)
```

Out[3]:

	SPEED	LON	LAT
2	22.6	24.696449	59.561600
3	22.5	24.715771	59.663120
4	22.4	24.744619	59.688480
5	22.7	24.794331	59.772049
6	22.5	24.716370	59.535412

Now, we need to check if the LON and LAT have a strong relation with SPEED or not. In this part, we compute the **Pearson correlation** of these three columns and plot it:

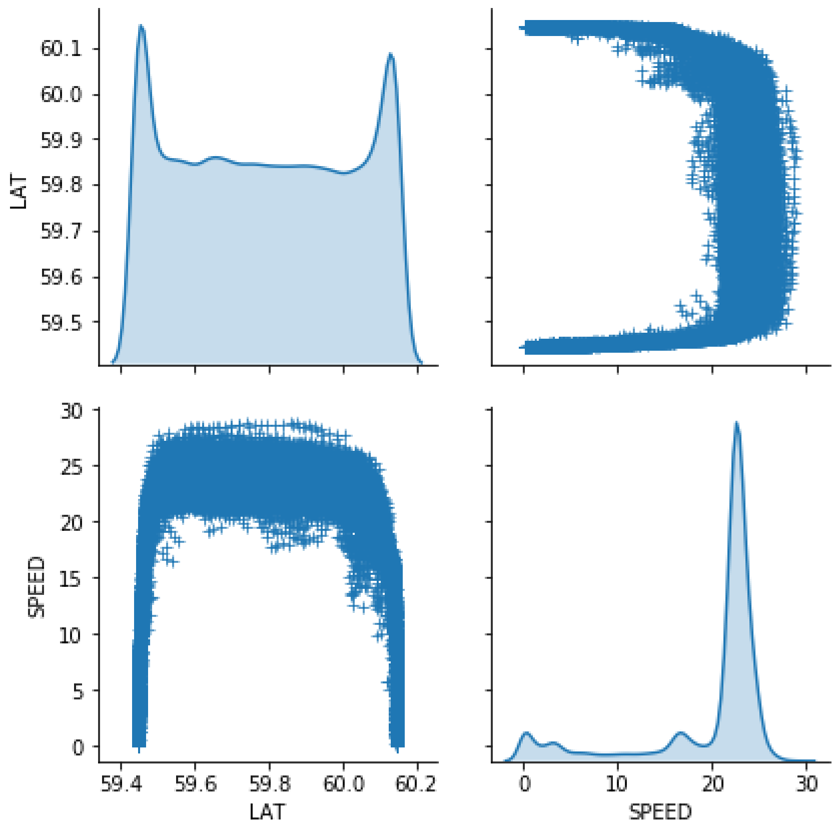
In [5]:
```python
import matplotlib.pyplot as plt
import seaborn as sns

def correlation(features):
    '''
    :param features: pd.dataframe
    :return: correlation matrix
    '''
    cor_matrix = features.corr(method="pearson")
    colormap = plt.cm.viridis
    plt.figure(figsize=(18, 12))
    plt.title('Pearson Correlation of Features', y=1.05, size=15)
    sns.heatmap(cor_matrix, linewidths=0.1, square=True, cmap=colormap, lineco
lor='white', annot=True)
    plt.show()
    return cor_matrix

cor_matrix = correlation(data)
```



From the figure above we could know that LAT and LON are very relevant to each other with a high correlation value 0.86, which meets our daily experiences. As for the correlation values of SPEED and LON, LAT, although they are not very high, can still be analyzed because they actually have a slight relationship. We can use a pair plot to show the correlation of two features more clearly:

Prior distribution:

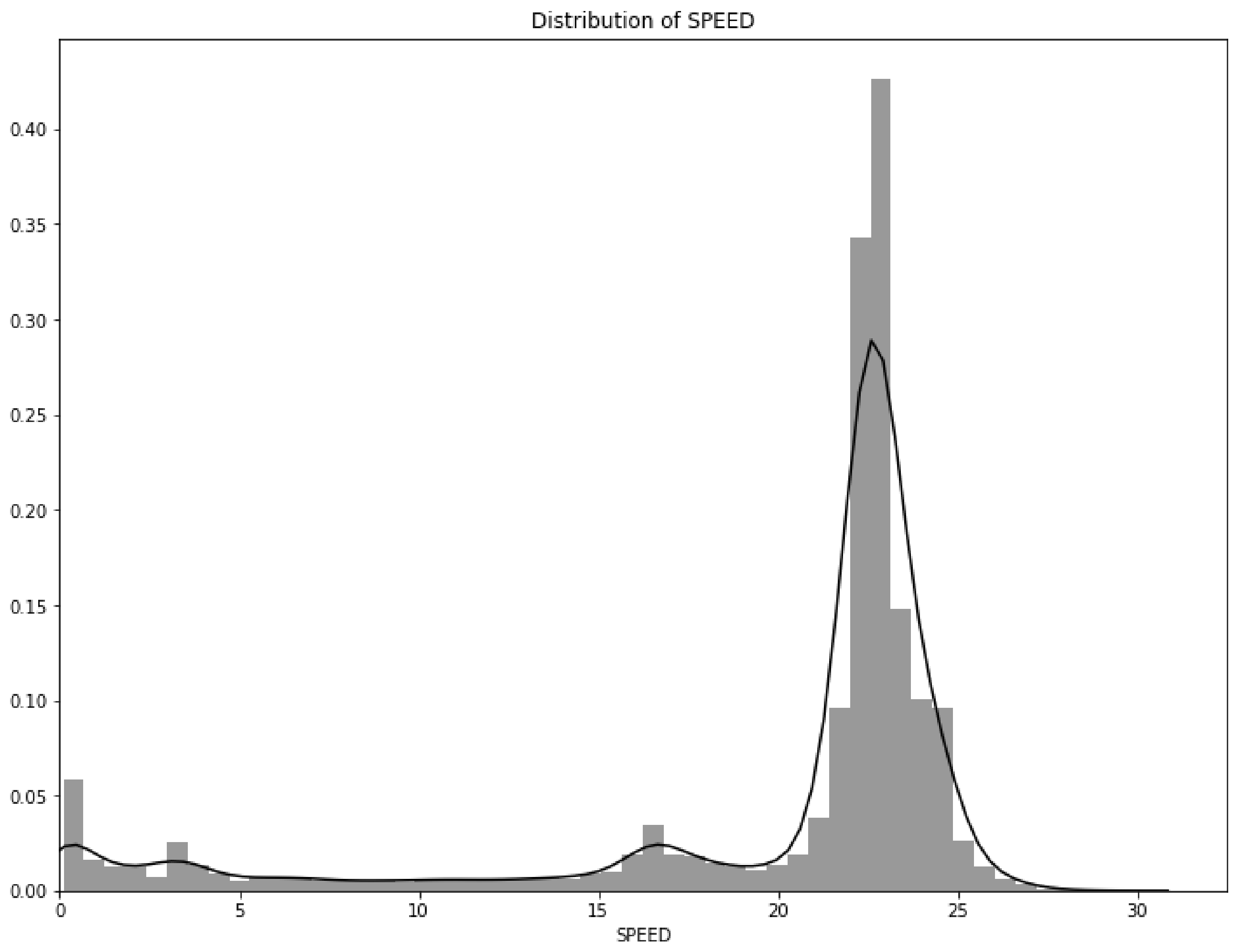
```
In [6]: def plot_pair(data, pair):
            sns.pairplot(data, height=3, vars=pair, markers="+", diag_kind="kde")
            plt.show()

        plot_pair(data, ('LAT', 'SPEED'))
```



From the scatter figure of LAT and SPEED, we can see a crude shape of normal distribution. It's really similar to the picture of normal distribution, so we can assume that our model of speed will probably a normal distribution.

Then, let us plot the prior distribution of SPEED, and check if it meets our assumptions:

In [7]:
```
def plot_prior(x):
    plt.figure(figsize=(12, 9))
    sns.distplot(x, bins=50, color='k', label=x.name)
    plt.title('Distribution of {}'.format(x.name))
    plt.xlim(left=0)
    plt.show()

plot_prior(data['SPEED'])
```



From the plot of SPEED, we could detect that its plot follows the shape of normal distribution in the intervals of [0,30] and [200, 280]. Furthermore, in the middle of the figure, it seems to follow a uniform distribution. Anyway, as this is only a project to test our knowledge in approximate bayesian Computation, so we don't need to pay lots time and attention to find a best model for our data. In this case, normal distribution for SPEED is neither a bad choice nor a good one, but let's us continue to do the task to see if our simulation of SPEED's posterior will make a success.

Model

Since we want to know the positions' effect to the speed of ship. The **prior** is the density of ship's move speed. Our observed data include the speed in the different latitude and longitude at the time. **Likelihood** in this case is pretty hard to compute, because we do not know the speed's effect to the current position, it's continuous and will be pretty hard to construct model with a large cost of computation sources. Therefore, it's an excellent alternative method to use MCMC to solve this problem approximately. Recently, with the progress of the research in Markov chain Monte Carlo (MCMC) especially the Hamiltonian Monte Carlo (HMC) method, we can get a pretty good simulation result.

We use Stan to implement our model. Stan is a state-of-the-art platform for statistical modeling and high-performance statistical computation. Thousands of users rely on Stan for statistical modeling, data analysis, and prediction in the social, biological, and physical sciences, engineering, and business.

Using PyStan package to call the Stan method in Python. PyStan provides an interface to Stan, a package for Bayesian inference using the No-U-Turn sampler, a variant of Hamiltonian Monte Carlo.

1. Non-Hierarchical Model:

This model is a simple linear model with gaussian noise, we combined the two paremeters of positions, LAT and LON to one new component, and named it as x. Speed of ship was represented as y, using gaussian model to simulate it.

We use four MC chains to sumulate our posterior, and each chain interate for 2000 times, 1000 for warm-up, left 1000 times for sampling. So for each model, we will have 4*1000 = 4000 samples.

In [18]:
```
import pystan
import numpy as np
from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import PCA


stan_code = '''
data{
    int<lower=0> N;
    vector[N] x;
    vector[N] y;
}

parameters{
    real alpha;
    real beta;
    real<lower=0> sigma;
}

model{
    y ~ normal(beta + alpha * x, sigma);
}

'''

std_data = StandardScaler().fit_transform(data[['LON', 'LAT']].values)
pca = PCA(n_components=1)
x = pca.fit_transform(std_data)
x = x.reshape(len(x))
y = data['SPEED'].values
model = pystan.StanModel(model_code=stan_code)
fit = model.sampling(data={'x': x[:200], 'y': y[:200], 'N': 200}, seed=16)

print(fit)
```

```
INFO:pystan:COMPILING THE C++ CODE FOR MODEL anon_model_321ef82d03545c8f52e0a
333c83ee8bd NOW.
/Users/jaywu/anaconda3/envs/master_courses/lib/python3.5/site-packages/pysta
n/misc.py:399: FutureWarning: Conversion of the second argument of issubdtype
from `float` to `np.floating` is deprecated. In future, it will be treated as
`np.float64 == np.dtype(float).type`.
  elif np.issubdtype(np.asarray(v).dtype, float):

Inference for Stan model: anon_model_321ef82d03545c8f52e0a333c83ee8bd.
4 chains, each with iter=2000; warmup=1000; thin=1;
post-warmup draws per chain=1000, total post-warmup draws=4000.

        mean se_mean     sd   2.5%    25%    50%    75%  97.5%  n_eff   Rhat
alpha  -0.76  5.8e-3   0.34  -1.44  -0.99  -0.75  -0.53  -0.09   3497    1.0
beta   19.72  7.5e-3   0.46  18.79  19.42  19.72  20.04  20.62   3804    1.0
sigma   6.38  5.1e-3   0.32   5.79   6.16   6.36   6.59   7.06   4000    1.0
lp__  -467.9    0.03    1.3 -471.3 -468.4 -467.5 -466.9 -466.4   2002    1.0

Samples were drawn using NUTS at Sun Dec  8 17:00:08 2019.
For each parameter, n_eff is a crude measure of effective sample size,
and Rhat is the potential scale reduction factor on split chains (at
convergence, Rhat=1).
```

2. Hierarchical Model:

We use the same data as have analysised in the non-hierarchical model before.

In [24]:
```
hierarchical_model = '''
data {
    int<lower=0> N;  // number of observations
    vector[N] x;
    vector[N] y;
}

transformed data {
    vector[N] x_std;
    vector[N] y_std;
    x_std = (x - mean(x)) / sd(x);
    y_std = (y - mean(y)) / sd(y);
}

parameters {
    real alpha;
    real beta;
    real<lower=0> sigma_std;
}

transformed parameters {
    vector[N] mu_std;
    mu_std = alpha + beta * x_std;
}

model {
    alpha ~ normal(0, 1);
    beta ~ normal(0,1);
    y_std ~ normal(mu_std, sigma_std);
}

generated quantities{
    vector[N] mu;
    real mu_mean;
    real<lower=0> sigma;
    mu = mu_std*sd(y) + mean(y);
    sigma = sigma_std * sd(y);
    mu_mean = mean(mu);
}
'''
model = pystan.StanModel(model_code=hierarchical_model)
fit = model.sampling(data={'x': x[:100], 'y': y[:100], 'N': 100}, seed=16)
print(fit.stansummary())
```

```
Inference for Stan model: anon_model_2a2ceed67ce8744e77562b11a13f9c6f.
4 chains, each with iter=2000; warmup=1000; thin=1;
post-warmup draws per chain=1000, total post-warmup draws=4000.

              mean se_mean     sd   2.5%     25%     50%     75%  97.5%  n_e
ff   Rhat
alpha      -3.5e-4  1.6e-3    0.1   -0.2   -0.07  2.1e-4    0.07    0.2   37
24    1.0
beta         -0.15  1.6e-3    0.1  -0.35   -0.22   -0.15   -0.08   0.05   37
68    1.0
sigma_std     1.01  1.1e-3   0.07   0.87    0.96     1.0    1.05   1.16   39
91    1.0
mu_std[1]     0.19  2.6e-3   0.16  -0.13    0.08     0.2     0.3    0.5   37
87    1.0
mu_std[2]     0.14  2.2e-3   0.14  -0.14    0.05    0.14    0.23    0.4   37
81    1.0
mu_std[3]      0.1  2.0e-3   0.12  -0.15    0.02     0.1    0.18   0.34   37
72    1.0
mu_std[4]     0.03  1.7e-3    0.1  -0.18   -0.04    0.03    0.09   0.22   37
37    1.0
mu_std[5]     0.18  2.5e-3   0.16  -0.13    0.08    0.18    0.29   0.48   37
86    1.0
mu_std[6]     0.12  2.1e-3   0.13  -0.14    0.03    0.12    0.21   0.37   37
77    1.0
mu_std[7]     0.06  1.8e-3   0.11  -0.16 -9.4e-3    0.06    0.13   0.27   37
56    1.0
mu_std[8]     0.04  1.7e-3    0.1  -0.17   -0.03    0.04    0.11   0.24   37
44    1.0
mu_std[9]  -9.0e-3  1.6e-3    0.1  -0.21   -0.07 -8.3e-3    0.06   0.19   36
93    1.0
mu_std[10]    0.16  2.4e-3   0.14  -0.14    0.06    0.16    0.25   0.43   37
84    1.0
mu_std[11]    0.15  2.3e-3   0.14  -0.14    0.06    0.15    0.24   0.42   37
82    1.0
mu_std[12]    0.15  2.3e-3   0.14  -0.14    0.06    0.15    0.25   0.42   37
83    1.0
mu_std[13]    0.17  2.4e-3   0.15  -0.13    0.07    0.17    0.27   0.45   37
85    1.0
mu_std[14]     0.2  2.7e-3   0.16  -0.13    0.09     0.2    0.31   0.51   37
87    1.0
mu_std[15]     0.2  2.7e-3   0.17  -0.13    0.09    0.21    0.32   0.52   37
88    1.0
mu_std[16]    0.07  1.8e-3   0.11  -0.15 -2.2e-4    0.07    0.15   0.29   37
61    1.0
mu_std[17]   -0.06  1.9e-3   0.11  -0.27   -0.13   -0.06  8.9e-3   0.15   33
10    1.0
mu_std[18]   -0.14  2.3e-3   0.13   -0.4   -0.22   -0.13   -0.05   0.12   32
34    1.0
mu_std[19]    0.16  2.4e-3   0.15  -0.14    0.07    0.16    0.26   0.44   37
84    1.0
mu_std[20]    0.17  2.5e-3   0.15  -0.13    0.07    0.18    0.28   0.47   37
86    1.0
mu_std[21]     0.2  2.8e-3   0.17  -0.13    0.09    0.21    0.32   0.53   37
88    1.0
mu_std[22]    0.08  1.9e-3   0.12  -0.15  9.4e-3    0.09    0.16    0.3   37
66    1.0
mu_std[23]   -0.03  1.7e-3    0.1  -0.23   -0.09   -0.03    0.04   0.17   35
56    1.0
mu_std[24]   -0.05  1.8e-3   0.11  -0.26   -0.12   -0.05    0.02   0.15   33
74    1.0
mu_std[25]   -0.07  1.9e-3   0.11  -0.29   -0.15   -0.07 -1.4e-4   0.14   32
64    1.0
mu_std[26]    -0.1  2.1e-3   0.12  -0.34   -0.18    -0.1   -0.02   0.13   32
26    1.0
mu_std[27]   -0.12  2.2e-3   0.13  -0.38   -0.21   -0.12   -0.04   0.13   32
```

```
25    1.0
mu_std[28]    -0.14  2.4e-3   0.14  -0.42   -0.24   -0.14   -0.05   0.12    32
44    1.0
mu_std[29]    -0.16  2.6e-3   0.15  -0.46   -0.26   -0.16   -0.07   0.12    32
69    1.0
mu_std[30]    -0.17  2.6e-3   0.15  -0.47   -0.27   -0.17   -0.07   0.12    32
79    1.0
mu_std[31]     0.15  2.3e-3   0.14  -0.14    0.06    0.15    0.25   0.42    37
83    1.0
mu_std[32]     0.19  2.6e-3   0.16  -0.13    0.08    0.19    0.29   0.49    37
87    1.0
mu_std[33]     0.17  2.5e-3   0.15  -0.13    0.07    0.18    0.28   0.46    37
86    1.0
mu_std[34]    -0.02  1.7e-3    0.1  -0.22   -0.08   -0.02    0.05   0.18    36
33    1.0
mu_std[35]    -0.04  1.7e-3    0.1  -0.24   -0.11   -0.04    0.03   0.17    34
69    1.0
mu_std[36]    -0.09  2.0e-3   0.12  -0.32   -0.17   -0.09   -0.01   0.14    32
36    1.0
mu_std[37]    -0.11  2.2e-3   0.12  -0.36   -0.19   -0.11   -0.03   0.13    32
23    1.0
mu_std[38]    -0.15  2.5e-3   0.14  -0.44   -0.25   -0.15   -0.06   0.12    32
56    1.0
mu_std[39]     0.16  2.4e-3   0.15  -0.14    0.06    0.16    0.26   0.44    37
84    1.0
mu_std[40]     0.05  1.7e-3   0.11  -0.16   -0.02    0.05    0.12   0.25    37
50    1.0
mu_std[41]    -0.16  2.5e-3   0.15  -0.45   -0.26   -0.16   -0.07   0.12    32
65    1.0
mu_std[42]    -0.18  2.7e-3   0.15  -0.48   -0.28   -0.18   -0.08   0.12    32
88    1.0
mu_std[43]    -0.19  2.8e-3   0.16  -0.51   -0.29   -0.19   -0.08   0.12    33
07    1.0
mu_std[44]     0.16  2.4e-3   0.14  -0.14    0.06    0.16    0.25   0.43    37
84    1.0
mu_std[45]      0.2  2.7e-3   0.17  -0.13    0.09     0.2    0.31   0.51    37
87    1.0
mu_std[46]     0.16  2.3e-3   0.14  -0.14    0.06    0.16    0.25   0.43    37
83    1.0
mu_std[47]    -0.21  3.0e-3   0.17  -0.55   -0.32   -0.21    -0.1   0.12    33
47    1.0
mu_std[48]    -0.22  3.0e-3   0.17  -0.56   -0.33   -0.21    -0.1   0.12    33
55    1.0
mu_std[49]    -0.21  2.9e-3   0.17  -0.54   -0.32   -0.21    -0.1   0.12    33
41    1.0
mu_std[50]     -0.2  2.8e-3   0.16  -0.52   -0.31    -0.2   -0.09   0.12    33
22    1.0
mu_std[51]    -0.19  2.7e-3   0.16   -0.5   -0.29   -0.19   -0.08   0.12    33
03    1.0
mu_std[52]    -0.03  1.7e-3    0.1  -0.23    -0.1   -0.03    0.03   0.17    35
09    1.0
mu_std[53]  -7.8e-3  1.6e-3    0.1  -0.21   -0.07 -7.0e-3    0.06   0.19    36
98    1.0
mu_std[54]     0.18  2.5e-3   0.16  -0.13    0.08    0.18    0.28   0.47    37
86    1.0
mu_std[55]     0.18  2.5e-3   0.16  -0.13    0.08    0.18    0.28   0.47    37
86    1.0
mu_std[56]     0.15  2.3e-3   0.14  -0.14    0.06    0.15    0.24   0.42    37
83    1.0
mu_std[57]     -0.2  2.8e-3   0.16  -0.53   -0.31    -0.2   -0.09   0.12    33
25    1.0
mu_std[58]    -0.21  2.9e-3   0.17  -0.55   -0.32   -0.21    -0.1   0.12    33
44    1.0
mu_std[59]    -0.16  2.6e-3   0.15  -0.46   -0.26   -0.16   -0.07   0.12    32
68    1.0
```

```
mu[90]         20.34    0.02    1.0  18.32   19.69   20.36   21.02  22.25   37
84    1.0
mu[91]         20.38    0.02   1.02  18.34   19.72   20.41   21.08  22.33   37
85    1.0
mu[92]          20.3    0.02   0.98  18.31   19.66   20.32   20.97  22.18   37
83    1.0
mu[93]         18.38    0.02   0.88   16.6    17.8   18.39   18.97  20.09   32
28    1.0
mu[94]         18.26    0.02   0.93  16.39   17.64   18.26   18.88  20.08   32
44    1.0
mu[95]         18.09    0.02   1.02  16.05   17.41   18.09   18.77  20.07   32
78    1.0
mu[96]         18.01    0.02   1.05  15.91   17.31   18.02   18.71  20.07   32
95    1.0
mu[97]         17.87    0.02   1.13  15.63   17.12   17.88   18.61  20.07   33
31    1.0
mu[98]         17.78    0.02   1.18  15.44    17.0    17.8   18.56  20.07   33
54    1.0
mu[99]          18.1    0.02   1.01  16.08   17.43    18.1   18.78  20.07   32
74    1.0
mu[100]        18.14    0.02   0.99  16.15   17.47   18.13   18.79  20.07   32
68    1.0
mu_mean        19.24    0.01   0.68  17.88    18.8   19.25   19.69  20.58   37
24    1.0
sigma           6.83  7.8e-3   0.49   5.94    6.49    6.79    7.14   7.89   39
91    1.0
lp__          -49.81    0.03   1.25 -53.09  -50.36  -49.49   -48.9  -48.4   19
12    1.0

Samples were drawn using NUTS at Sun Dec  8 18:17:26 2019.
For each parameter, n_eff is a crude measure of effective sample size,
and Rhat is the potential scale reduction factor on split chains (at
convergence, Rhat=1).
```

Checking $\hat{R}$:

The first thing we need to check after we get the samples is if $\hat{R}$ is close to 1. From the detailed information of fit above, we can find that our result of $\hat{R}$ is fine, they all close to 1, which means our simulation is convergent.

Checking Effective Sample Size(ESS):

We can see the n_eff above, the issue here is related to the fact that we are estimating the effective sample size from the fit output. When n_eff / n_transitions < 0.001 the estimators that we use are often biased and can significantly overestimate the true effective sample size. In our case here, the value of ESS is much bigger than the interval of wrong values, which means our model is suitable.

Checking Divergences, Tree Depth, and other Diagnostics:

We can check divergences which indicate pathological neighborhoods of the posterior that the simulated Hamiltonian trajectories are not able to explore sufficiently well.

The dynamic implementation of HMC in Stan has a maximum trajectory length built in to avoid infinite loops that can occur for non-identified models

We can use the **stan_utility** module to check all the diagnostics quickly and conveniently:

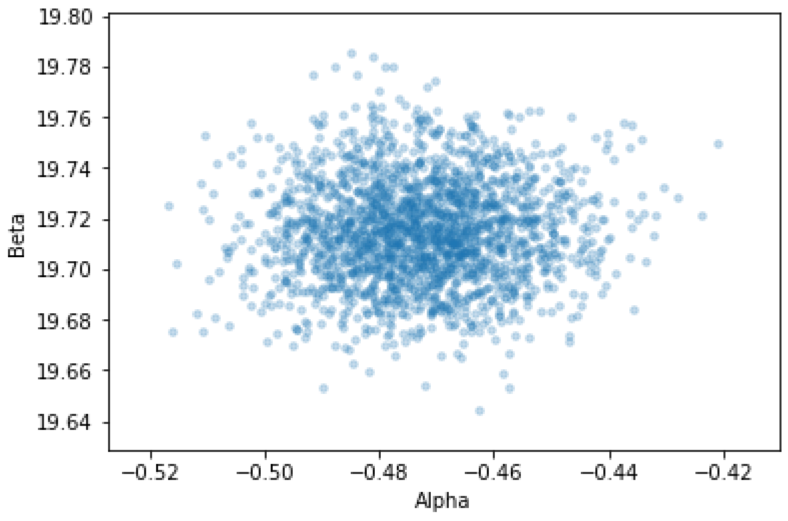
In [15]:
```
import stan_utility
stan_utility.check_all_diagnostics(fit)
```

```
n_eff / iter looks reasonable for all parameters
Rhat looks reasonable for all parameters
0.0 of 4000 iterations ended with a divergence (0.0%)
8 of 4000 iterations saturated the maximum tree depth of 10 (0.2%)
  Run again with max_treedepth set to a larger value to avoid saturation
E-BFMI indicated no pathological behavior
```

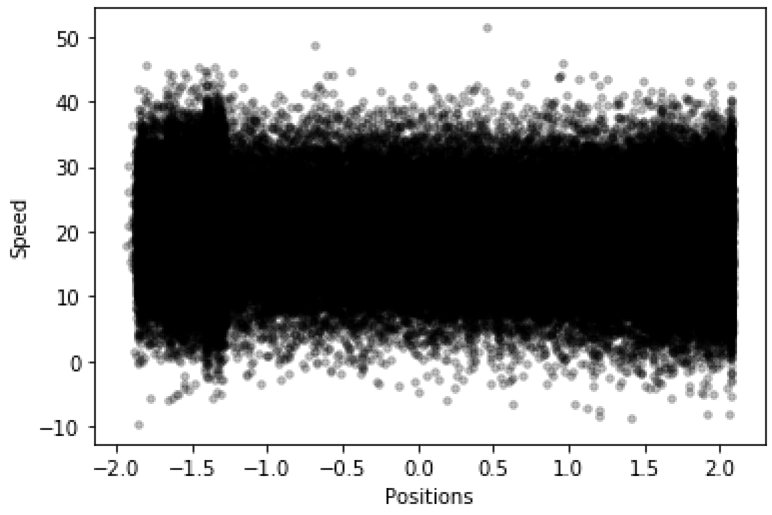
Posterior Predictive Checking:

The values of Rhat and n_eff both show that our simulation get a convergence and perform well. Now, let us plot the posterior density.

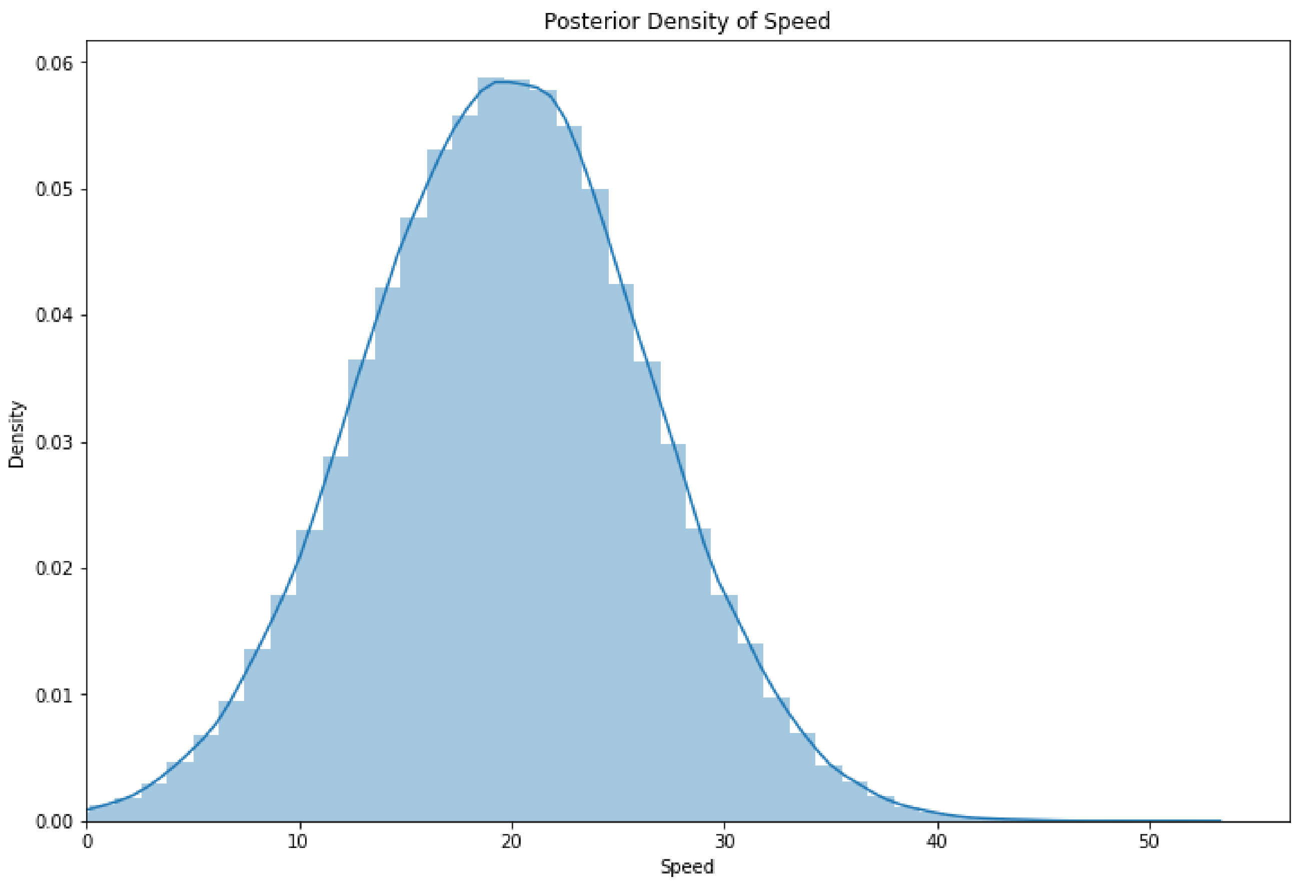
In [17]:
```
import pickle
import pystan

def read_model(model_name='model_fit.pkl'):
    with open(model_name, "rb") as f:
        data_dict = pickle.load(f)
    fit = data_dict['fit']
    model = data_dict['model']
    return fit, model

fit, model = read_model()
samples = fit.extract(permuted=True)
means = fit.get_posterior_mean()
alpha = round(means[0].mean(), 2)
beta = round(means[1].mean(), 2)
sigma = round(means[2].mean(), 2)
plt.scatter(samples['alpha'][:2000], samples['beta'][:2000], 50, marker='.', a
lpha=0.25)
plt.xlabel('Alpha')
plt.ylabel('Beta')
plt.show()
```



In [18]:
```
def plot_density(alpha, beta, sigma, x):
    y = np.ndarray(len(x))
    for i, k in enumerate(x):
        y[i] = np.random.normal(alpha * x[i] + beta, sigma)
    plt.scatter(x, y, 50, marker='.', alpha=0.25, color='k')
    plt.xlabel('Positions')
    plt.ylabel('Speed')
    plt.show()
    return y

y_sample = plot_density(alpha, beta, sigma, x)
```



In [20]:
```
def plot_density_feature(f):
    plt.figure(figsize=(12, 8))
    plt.tight_layout()
    sns.distplot(f, hist=True, kde=True)
    plt.xlabel('Speed')
    plt.ylabel('Density')
    plt.xlim(left=0)
    plt.title('Posterior Density of Speed')
    plt.show()

plot_density_feature(y_sample)
```



Discussion and Potential Improvements

From the result we could know that the speed of a ship obeys the normal distribution, and the most frequent speed value is between [10, 30]. Although the relationship between the speed and position is not very strong, we can still use the result of this project to optimize the best sail route for a ship in the future work. In this project, we only use the one-dimension normal distribution to simulate the practical density of ship's speed. This way is easy to implement and will save lots of time since we use PCA method to reduce the dimensionality of parameters, that is LAT and LON. Maybe in the future section, we will add a bivariate normal model to represent the change of speed with the relations of positions.

Thanks for reading!