Homework 5

November 30, 2015

1 Inference Homework 5

1.1 Alex Pine

1.2 2015/11/21

1.3 Question 1

I collaborated with Israel Malkin and Peter Li on this problem.

Let T be a tree MRF, and let T' be a directed tree with the same skeleton as T, and one root node, R, which has all edges are directed away from it.

T' must be a moral graph, where each node has at most one parent. If it were not, that would mean there is some node n with at least two parents, p_1 and p_2. Then there would be at least two directed paths which end with n, one including p_1, and another including p_2. If both of those paths started with R, T would not be a tree, because trees have exactly one path between two nodes by definition. If instead, one of the paths ends in a node that is not R, this node would constitute a second root node, which does not exist by construction. Thus T' must be moral.

Since T' is moral, it encodes the same probability distribution as T, which we'll call $p_T(\mathbf{x})$. $p_T(\mathbf{x})$ can thus be factorized according to T':

$$p_T(\mathbf{x}) = \prod_{x_i \in \mathbf{x}} p(x_i \mid \mathrm{pa}(x))$$

Since all nodes have at most one parent node, we can denote $\mathrm{pa}(x) = x_p$.

$$p_T(\mathbf{x}) = \prod_{x_i \in \mathbf{x}} p(x_i \mid x_p)$$

Applying Bayes' rule, this becomes

$$p_T(\mathbf{x}) = \prod_{x_i \in \mathbf{x}} \frac{p(x_i, x_p)}{p(x_p)}$$

Multiplying this by $\prod_{x_i \in \mathbf{x}} \frac{p(x_i)}{p(x_i)} = 1$, this becomes

$$p_T(\mathbf{x}) = \prod_{x_i \in \mathbf{x}} \frac{p(x_i, x_p)}{p(x_i)p(x_p)} \prod_{x_i \in \mathbf{x}} p(x_i)$$

Since the set of edges T' are between parents and children, and each node is a child of one or zero parents, each i appears once in the edges $(i, j) \in T'$. This means we can change the indices of the first product to be the set of edges in T' without changing the value of the expression. If we also denote V to be the set containing the marginal variables of $\mathbf{x}$, then this formula can be rewritten as

$$p_T(\mathbf{x}) = \prod_{(i,j) \in T} \frac{p(x_i, x_j)}{p(x_i)p(x_j)} \prod_{x_i \in V} p(x_i)$$

This probability distribution corresponds to an MRF with the same nodes and edges as T, where each node x_i has the potential $p(x_i)$, and each edge has the potential $p(x_i, x_j)$, which is what we were trying to prove.

2 Question 2

I collaborated with Maya Rotmensch, Peter Li, and Israel Malkin on this problem.

The output was written to a file called "hw5_prob2.uai". Some debugging output is shown at the end of the code to illustrate that it works properly.

```
In [3]: import numpy as np

        def read_chowlui_input():
            input_matrix = []
            with open('inference/hw5/data/chowliu-input.txt', 'r') as f:
                for line in f:
                    input_matrix.append([int(token) for token in line.split()])
            return np.array(input_matrix)

        def read_labels_input():
            input_labels = []
            with open('inference/hw5/data/names.txt', 'r') as f:
                for line in f:
                    input_labels.append(line.strip())
            return np.array(input_labels)

        input_matrix = read_chowlui_input()
        labels = read_labels_input()
```

```
In [4]: import scipy.sparse.csgraph

        # Given the input matrix, which has one row for each data point, and a column for each item,
        # create two matrices:
        # 1) both_count_matrix:
        #    When i > j: A[i][j] = the number of times items i and j were both present.
        #    When i == j: A[i][j] = the number of times item i appears in the data set.
        #    When i < j: A[i][j] = zero. This portion is not used.
        # 2) single_count_matrix:
        #    When i != j: B[i][j] = the number of times item i was present and j was not.
        #    When i == j: B[i][j] = zero. This portion is not used.
        def create_count_matrices(input_matrix):
            both_count_matrix = np.zeros((input_matrix.shape[1], input_matrix.shape[1]), dtype=int)
            single_count_matrix = np.zeros((input_matrix.shape[1], input_matrix.shape[1]), dtype=int)
            for row in input_matrix:
                for i in range(len(row)):
                    for j in range(len(row)):
                        if i > j and row[i] == 1 and row[j] == 1:
                            both_count_matrix[i][j] += 1
                        if row[i] == 1 and row[j] == 0:
                            single_count_matrix[i][j] += 1
                    if row[i] == 1:
                        both_count_matrix[i][i] += 1
            return both_count_matrix, single_count_matrix

        both_count_matrix, single_count_matrix = create_count_matrices(input_matrix)
```

In [6]:
```python
# Functions needed to create a matrix that has the mutual information between each item,
# based on their empirical probabilities.
# I actually take the inverse of the mutual information, so that a minimum spanning tree
# algorithm can be used to find the maximum spanning tree.

import itertools
import math
import numpy.ma as ma

# Computes one part of the sum of the "empirical" mutual information between x_i and x_j.
def mutual_info_part(num_xixj, num_xi, num_xj, total):
    if not num_xixj or not num_xi or not num_xj:
        return 0.0
    p_xixj = 1.0 * num_xixj / total
    p_xi = 1.0 * num_xi / total
    p_xj = 1.0 * num_xj / total
    return p_xixj * math.log(p_xixj / (p_xi * p_xj), 2)


def mutual_info(i, j, num_data, both_count_matrix, single_count_matrix):
    num_xi_one = both_count_matrix[i][i]
    num_xj_one = both_count_matrix[j][j]
    num_xi_zero = num_data - num_xi_one
    num_xj_zero = num_data - num_xj_one
    info = 0.0
    # 11 case
    info += mutual_info_part(both_count_matrix[i][j], num_xi_one, num_xj_one, num_data)
    # 10 case
    info += mutual_info_part(single_count_matrix[i][j], num_xi_one, num_xj_zero, num_data)
    # 01 case
    info += mutual_info_part(single_count_matrix[j][i], num_xi_zero, num_xj_one, num_data)
    # 00 case
    neither_count = (num_data - both_count_matrix[i][j] - single_count_matrix[i][j]
                     - single_count_matrix[j][i])
    info += mutual_info_part(neither_count, num_xi_zero, num_xj_zero, num_data)
    assert (neither_count + both_count_matrix[i][j] + single_count_matrix[i][j]
            + single_count_matrix[j][i] == num_data)
    return info

# Create a matrix where each value is the inverse of the mutual information values
# between each item.
def create_inv_mutual_info_matrix(num_data, both_count_matrix, single_count_matrix):
    inv_info_matrix = np.zeros(both_count_matrix.shape)
    # Create a mask for the upper right corner, to mark this matrix as triangular.
    info_mask = np.zeros(both_count_matrix.shape, dtype=int)
    num_items = both_count_matrix.shape[1]
    for i in range(num_items):
        for j in range(i):
            info = mutual_info(i, j, num_data, both_count_matrix, single_count_matrix)
            inv_info_matrix[i][j] = 1.0 / info
        for j in range(i, num_items):
            info_mask[i][j] = 1
    inv_info_matrix = ma.array(inv_info_matrix, mask=info_mask)
    return inv_info_matrix
```

```
    num_data = input_matrix.shape[0]
    inv_info_matrix = create_inv_mutual_info_matrix(
        num_data, both_count_matrix, single_count_matrix)
```

In [7]:
```
import scipy.sparse.csgraph

mst_rows, mst_columns = scipy.sparse.csgraph.minimum_spanning_tree(inv_info_matrix).nonzero()
mst_edge_pairs = zip(mst_rows, mst_columns)
```

In [8]:
```
# Code for calculating the potential functions on nodes and edges, now that we have the
# structure of the tree.

def edge_part(num_xixj, num_xi, num_xj, total):
    p_xixj = 1.0 * num_xixj / total
    p_xi = 1.0 * num_xi / total
    p_xj = 1.0 * num_xj / total
    return p_xixj / (p_xi * p_xj)


def edge_potential(i, j, num_data, both_count_matrix, single_count_matrix):
    num_xi_one = both_count_matrix[i][i]
    num_xj_one = both_count_matrix[j][j]
    num_xi_zero = num_data - num_xi_one
    num_xj_zero = num_data - num_xj_one
    potential = np.zeros((2, 2))
    # 11 case
    potential[1][1] = edge_part(both_count_matrix[i][j], num_xi_one, num_xj_one, num_data)
    # 10 case
    potential[1][0] = edge_part(single_count_matrix[i][j], num_xi_one, num_xj_zero, num_data)
    # 01 case
    potential[0][1] = edge_part(single_count_matrix[j][i], num_xi_zero, num_xj_one, num_data)
    # 00 case
    neither_count = (num_data - both_count_matrix[i][j] - single_count_matrix[i][j]
                     - single_count_matrix[j][i])
    potential[0][0] = edge_part(neither_count, num_xi_zero, num_xj_zero, num_data)
    assert (neither_count + both_count_matrix[i][j] + single_count_matrix[i][j]
            + single_count_matrix[j][i] == num_data)
    return potential


def node_potential(i, num_data, both_count_matrix):
    num_xi_one = both_count_matrix[i][i]
    num_xi_zero = num_data - num_xi_one
    return np.array([float(num_xi_one) / num_data, float(num_xi_zero) / num_data])


def create_graph_potentials(mst_edge_pairs, num_data,
                            both_count_matrix, single_count_matrix):
    # Every node i has a 1x2 potential matrix
    node_potentials = np.ndarray((both_count_matrix.shape[0], 2))
    # Every edge (i,j) has a 2x2 potential matrix.
    edge_potentials = np.ndarray((both_count_matrix.shape[0], both_count_matrix.shape[1],
                                  2, 2))
```

```
    num_items = both_count_matrix.shape[1]
    for i in range(num_items):
        for j in range(i):
            edge_potentials[i][j] = edge_potential(i, j, num_data, both_count_matrix,
                                                   single_count_matrix)
        node_potentials[i] = node_potential(i, num_data, both_count_matrix)
    return node_potentials, edge_potentials

node_potentials, edge_potentials = create_graph_potentials(
    mst_edge_pairs, num_data, both_count_matrix, single_count_matrix)
```

In [10]:
```
# Creates a string that reprsents the Markov random field in UAI format.
# http://www.hlt.utdallas.edu/~vgogate/uai14-competition/modelformat.html
def create_graph_as_uai_file(mst_edge_pairs, node_potentials, edge_potentials):
    num_vars = node_potentials.shape[0]

    network_type = 'MARKOV'
    num_vars_str = str(num_vars)
    var_cardinals = ' '.join(['2']*num_vars)
    num_cliques = str(len(mst_edge_pairs) + num_vars)
    node_cliques = ['1 ' + str(i) for i in range(num_vars)]
    edge_cliques = [' '.join(['2', str(i), str(j)]) for i, j in mst_edge_pairs]
    preamble = ([network_type, num_vars_str, var_cardinals, num_cliques] + node_cliques
                + edge_cliques)

    function_tables = []
    # node potentials
    for i in range(num_vars):
        prob_str = ' '.join([' ', str(node_potentials[i][0]), str(node_potentials[i][1])])
        function_tables += ['', '2', prob_str]
    # edge potentials
    for i, j in mst_edge_pairs:
        assert i > j
        prob00 = edge_potentials[i][j][0][0]
        prob01 = edge_potentials[i][j][0][1]
        prob_str1 = ' '.join([' ', str(prob00), str(prob01)])
        prob10 = edge_potentials[i][j][1][0]
        prob11 = edge_potentials[i][j][1][1]
        prob_str2 = ' '.join([' ', str(prob10), str(prob11)])
        function_tables += ['', '4', prob_str1, prob_str2]
    return '\n'.join(preamble + function_tables + [''])

uai_str = create_graph_as_uai_file(mst_edge_pairs, node_potentials, edge_potentials)

with open ('hw5_prob2.uai', 'w') as f:
    f.write(uai_str)
```

In [15]:
```
# Debug output - Edge potentials when both items are present for the first 25 edges.
# Format:
# item 1 index, item 2 index, item 1 label -- item 2 label : edge potential when
# both are present.

for i, j in mst_edge_pairs[:25]:
    print i, j, labels[i], '--', labels[j], ":", edge_potentials[i][j][1][1]
```

```
13 11 books -- book : 17.8188951664
13 12 books -- bookcase : 32.2502421308
15 14 bottles -- bottle : 21.8532110092
18 17 boxes -- box : 13.7630003152
20 2 building -- awning : 2.83260887878
20 4 building -- balcony : 3.15318242536
22 16 cabinet -- bowl : 15.1463651498
25 24 cars -- car : 7.9865994979
32 22 countertop -- cabinet : 19.6495829675
35 8 cushion -- bed : 12.1792726461
36 26 desk -- chair : 5.25427127617
36 28 desk -- clock : 6.07160236357
39 20 dome -- building : 2.81421531463
40 6 door -- bars : 3.19225146199
41 22 drawer -- cabinet : 15.4129411765
45 1 fireplace -- armchair : 15.1631944444
46 1 floor -- armchair : 2.60851983636
46 13 floor -- books : 2.42753607988
46 20 floor -- building : 0.0921954638603
46 26 floor -- chair : 2.34104808146
46 27 floor -- chandelier : 2.52800622804
47 23 flowers -- candle : 12.0635359116
51 46 ground -- floor : 0.0828942173358
52 40 handrail -- door : 2.77395644283
53 50 headstone -- grass : 7.11527494908
```

2.1 Question 3

I collaborated with Israel Malkin and Peter Li on this problem.

Define kernel function $p(x_i, x_j) = C^{-1}_{i,j}$, where C is a covariance matrix and $C_{i,j} = k(x_i, x_j)$, and $x_i, x_j \in \mathbf{x_n}$, a sequence of i.i.d random variables.

Drawing the first variable, x_1, results in a covariance matrix $C_1 = \begin{pmatrix} k(x_1, x_1) \end{pmatrix}$, which implies that $C_1^{-1} = \begin{pmatrix} \frac{1}{k(x_1,x_1)} \end{pmatrix}$ which implies $p(x_1, x_1) = \frac{1}{k(x_1,x_1)}$.

Drawing a second variable, x_2, results in a covariance matrix

$$C_2 = \begin{pmatrix} k(x_1, x_1) & k(x_1, x_2) \\ k(x_1, x_2) & k(x_2, x_2) \end{pmatrix}$$

Since C_2 is a covariance matrix, it must be symmetric, so $k(x_1, x_2) = k(x_2, x_1)$, and its inverse is given by

$$C_2^{-1} = \frac{1}{k(x_1, x_1)k(x_2, x_2) - k(x_1, x_2)^2} \begin{pmatrix} k(x_2, x_2) & -k(x_1, x_2) \\ -k(x_1, x_2) & k(x_1, x_1) \end{pmatrix}$$

Since the first element of $C_2^{-1} = p(x_1, x_1)$, this implies

$$p(x_1, x_1) = \frac{k(x_2, x_2)}{k(x_1, x_1)k(x_2, x_2) - k(x_1, x_2)^2}$$

Plugging in $p(x_1, x_1) = \frac{1}{k(x_1,x_1)}$ from above, this turns into

$$\frac{1}{k(x_1, x_1)} = \frac{k(x_2, x_2)}{k(x_1, x_1)k(x_2, x_2) - k(x_1, x_2)^2}$$

Rearranging, this becomes,

$$k(x_1, x_1)k(x_2, x_2) = k(x_1, x_1)k(x_2, x_2) - k(x_1, x_2)^2$$

which implies

$$k(x_1, x_2) = k(x_2, x_1) = 0$$

This means that $p(x_i, x_j)$ only exists if C is a diagonal matrix. Since C is not required to be diagonal, this implies that $p(x_i, x_j)$ does not exist generally.

2.2 Question 4

I collaborated with Peter Li on this problem.

I've graphed four covariance functions:

1) Exponential (not differentiable, not compact)
2) Squared Exponential (differentiable, not compact)
3) Spherical (not differentiable, compact)
4) Linear (differentiable, not compact)

In [435]:
```
# Covariance functions

import itertools
import math
import matplotlib.pyplot as plt
import numpy as np

from mpl_toolkits.mplot3d import Axes3D
from matplotlib import cm

%matplotlib inline

# 1) Exponential (not differentiable, not compact)
def make_exp_covar_fn(tau, l):
    def exp_covar_fn(x, y):
        return tau**2 * math.exp(-np.linalg.norm(x - y) / (2*l))
    return exp_covar_fn

# 2) Squared Exponential (differentiable, not compact)
def make_squared_exp_covar_fn(tau, l):
    def squared_exp_covar_fn(x, y):
        diff = x - y
        return tau**2 * math.exp(-np.inner(diff, diff) / (2*(l**2)))
    return squared_exp_covar_fn

# 3) Spherical (not differentiable, compact)
def make_spherical_covar_fn(tau, theta):
    def spherical_covar_fn(x, y):
        dist = np.linalg.norm(x - y)
        if dist <= theta:
            return tau**2 * (1 - 3*dist/(2*theta) + dist**3/(2*theta**3))
        else:
            return 0.0
    return spherical_covar_fn

# 4) Linear (differentiable, not compact)
```

```
def make_linear_covar_fn(sigma, tau, c):
    def linear_covar_fn(x, y):
        return sigma**2 + tau**2 * np.inner(x - c, y - c)
    return linear_covar_fn
```

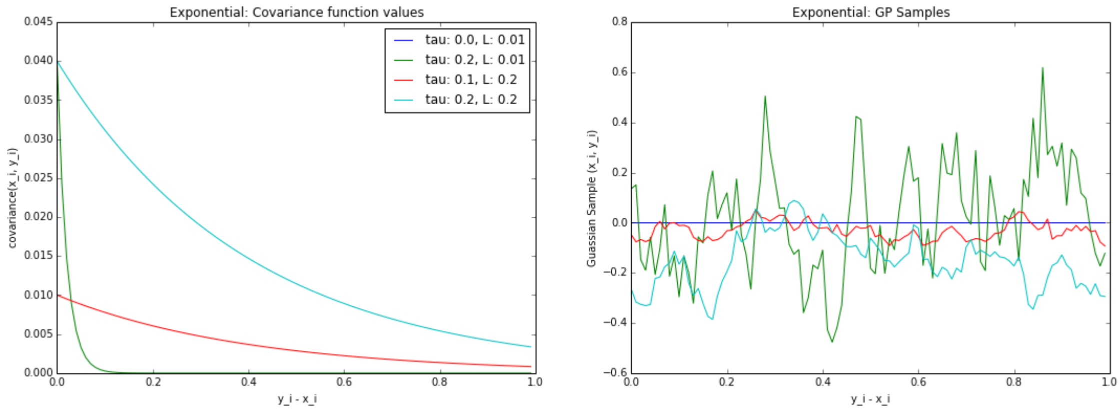
In [317]:
```
# This samples from a multivariate Gaussian distribution, with a zero mean
# value, and a covariance matrix determined by the convariance function
# and the x values parameters.
def sample_gp(xs, covar_fn):
    mean_vec = np.zeros(len(xs))
    gram_matrix = np.zeros((len(xs), len(xs)))
    for i in range(len(xs)):
        x_i = xs[i]
        for j in range(0, i):
            x_j = xs[j]
            gram_matrix[i][j] = covar_fn(x_i, x_j)
            gram_matrix[j][i] = gram_matrix[i][j]
        gram_matrix[i][i] = covar_fn(x_i, x_i)
    samples = np.random.multivariate_normal(mean_vec, gram_matrix)
    return samples
```

In [491]:
```
def plot_covar_fn(title, covar_fn_maker, param_list, param_names):
    covar_fn_list = [covar_fn_maker(*params) for params in param_list]
    ys = np.arange(0.0, 1.0, 0.01)

    covar_vals = []
    sample_vals = []
    for covar_fn in covar_fn_list:
        covar_vals.append([covar_fn(0.0, y) for y in ys])
        sample_vals.append(sample_gp(ys, covar_fn))

    fig = plt.figure(1, figsize=(18, 6))
    # Covariance values
    plt.subplot(1, 2, 1)
    plt.title(title + ': Covariance function values')
    plt.xlabel('y_i - x_i')
    plt.ylabel('covariance(x_i, y_i)')
    lines = []
    for params, covar_val in zip(param_list, covar_vals):
        param_str = ", ".join([name + ": " + str(param)
                               for name, param in zip(param_names, params)])
        line, = plt.plot(ys, covar_val)
        lines.append(param_str)
    plt.legend(lines)
    # Gaussian Process samples
    plt.subplot(1, 2, 2)
    plt.title(title + ': GP Samples')
    for sample_val in sample_vals:
        plt.plot(ys, sample_val)
    plt.xlabel('y_i - x_i')
    plt.ylabel('Guassian Sample (x_i, y_i)')
    plt.show()
```

In [492]:
```
# L can't be zero or you get a divide by zero error
```

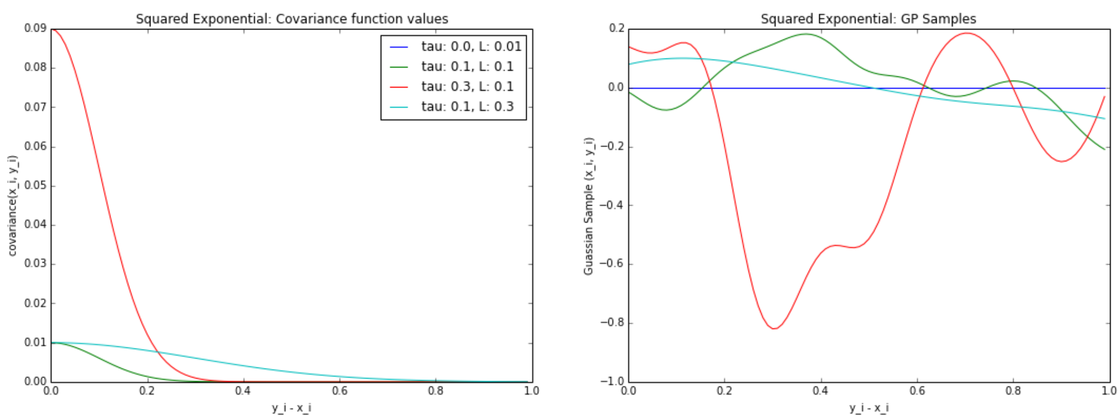
```
exp_params = [(0.0, 0.01), (0.2, 0.01), (0.1, 0.2), (0.2, 0.2)]
plot_covar_fn("Exponential", make_exp_covar_fn, exp_params, ['tau', 'L'])
```

```
/Library/Python/2.7/site-packages/ipykernel/__main__.py:13: DeprecationWarning: using a non-integer numbe
```



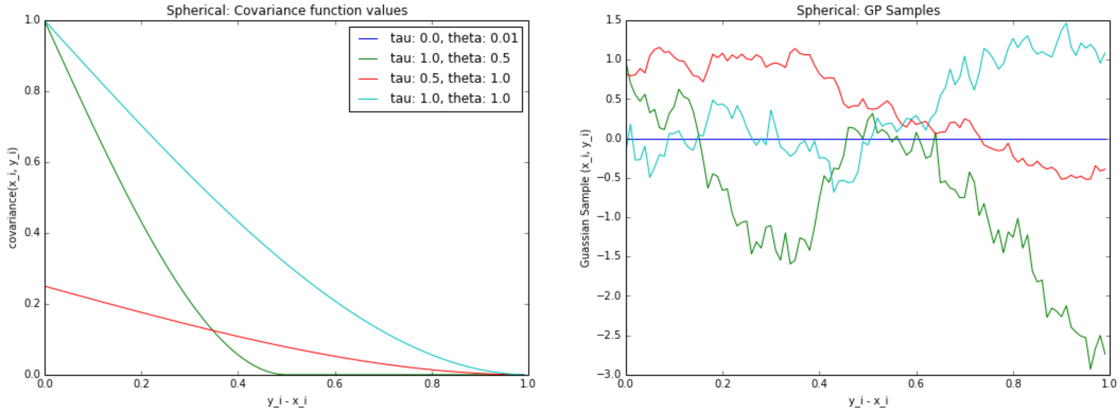

```
In [494]: squared_exp_params = [(0.0, 0.01), (0.1, 0.1), (0.3, 0.1), (0.1, 0.3)]
          plot_covar_fn("Squared Exponential", make_squared_exp_covar_fn,
                        squared_exp_params, ['tau', 'L'])
```

```
/Library/Python/2.7/site-packages/ipykernel/__main__.py:13: DeprecationWarning: using a non-integer numbe
```




```
In [495]: spherical_params = [(0.0, 0.01), (1.0, 0.5), (0.5, 1.0), (1.0, 1.0)]
          plot_covar_fn("Spherical", make_spherical_covar_fn,
                        spherical_params, ['tau', 'theta'])
```

```
/Library/Python/2.7/site-packages/ipykernel/__main__.py:13: DeprecationWarning: using a non-integer numbe
```




```
In [496]: linear_param_list = [(0.0, 0.0, 0.0), (0.0, 2.0, 2.0), (1.0, 2.0, 2.0), (2.0, 1.0, 1.0)]
          plot_covar_fn("Linear", make_linear_covar_fn, linear_param_list, ['sigma', 'tau', 'c'])
```

```
/Library/Python/2.7/site-packages/ipykernel/__main__.py:13: DeprecationWarning: using a non-integer numbe
```