Part II, Question 1

Code:

```
# Initialize shape of both matrices
row1, col1   # Shape of M
row2, col2   # Shape of N

def map(k):
   pairs = []
   matrix_name = k[0]
   i = k[1][0][0]
   j = k[1][0][1]
   v = k[1][1]

   if Matrix == "M":
      for k in range(col2):
         pairs.append(((i, k), [(matrix_name, j, v)]))      # ((i, k), (M, j, Mij))
   else:
      for k in range(row1):
         pairs.append(((k, j), [(matrix_name, i, v)]))      # ((k, j), (N, i, Nij))

   return pairs

def reduceByKey(a, b):
   return (a + b)

def reduce(vs):
   key = vs[0]
   value_list = vs[1]
   list_M = [0] * col1
   list_N = [0] * col1

   for (matrix_name, j, v) in value_list:
      if matrix_name == 'M':
         list_M[j] = v
      else:
         list_N[j] = v

   sum = 0
   for i in range(col1):
      sum = sum + list_M[i] * list_N[i]

   return (key, sum)

# Function for each RDD of the form: (matrix_name, ((i, j), v))
def MatrixMultiply_Spark(rdd):
   pairs_after_map = rdd.flatMap(lambda x: map(x))
   pairs_reduced_by_key = pairs_after_map.reduceByKey(lambda x: reduceByKey(x))
   pairs_after_reduce = pairs_reduced_by_key.map(lambda x: reduce(x))

   return pairs_after_reduce
```

Assumptions:

1. Shape of both matrices are initialized initially before calling MatrixMultiply_Spark function for each RDD and are accessible globally.
   Shape of Matrix M: (row1, col1)
   Shape of Matrix N: (row2, col2)

Description of Spark Transformations:

1. **FlatMap**
   This maps each rdd of the form (matrix_name, ((i, j), v)) to key-values pairs where:
   For matrix_name = 'M': Key-value pair: ((i, k), [('M', j, M_{ij})]) where k = [0, col2)
   For matrix_name = 'N': Key-value pair: ((k, j), [('N', i, N_{ij})]) where k = [0, row1)

2. **ReduceByKey**
   Reduces all (Key, [Value]) pairs having same Key to (Key, [$Value_1$, $Value_2$, …, $Value_N$]).

3. **Map**
   Gets input as (Key, [$Value_1$, $Value_2$, ...$Value_N$]) where $Value_i$ is of the type: (matrix_name, j, v)
   The reduce function called within the map does the following:
   a. Multiplies v of the two tuples which have same value of j
   b. Sums up result of all multiplications
   c. Returns result as (Key, Sum)
   Here Key = (i, k) i.e. the (row_no, col_no) of the resulting matrix and Sum is the final value at that position.

Part II, Question 2

Code:

```
def testL2Reg(penalty_value = 1.0, learning_rate = 0.0001, n_epochs = 100, momentum = 0.9):
    # features and outcomes:
    X = tf.constant(featuresZ_pBias, dtype=tf.float32, name="X")
    y = tf.constant(price.reshape(-1,1), dtype=tf.float32, name="y")

    # weight/parameters
    beta = tf.Variable(tf.random_uniform([featuresZ_pBias.shape[1], 1], -1., 1.), name = "beta")
    # m is set to a variable having same shape as beta and initialized to 0.
    m = tf.Variable(tf.zeros([featuresZ_pBias.shape[1], 1]), name = "m")

    # the linear model:
    y_pred = tf.matmul(X, beta, name="predictions")

    # setup the cost function:
    penalty = tf.constant(penalty_value, dtype=tf.float32, name="penalty")
    penalizedCost = tf.reduce_sum(tf.square(y - y_pred)) + penalty * tf.reduce_sum(tf.square(beta))

    # add in gradient calculation
    grads = tf.gradients(penalizedCost, [beta])[0]

    # add gradient descent to graph using momentum optimization
    momentum_vector = tf.assign(m, ((momentum*m)+(learning_rate*grads)) )
    training_op = tf.assign(beta, beta-m)

    #run the graph
    init = tf.global_variables_initializer()
    with tf.Session() as sess:
        sess.run(init)
        for epoch in range(n_epochs):
            if epoch %10 == 0: #print debugging output
                print("Epoch", epoch, "; penalizedCost =", penalizedCost.eval())
            # For each iteration: Find new value of momentum vector (m) and beta
            sess.run(momentum_vector)
            sess.run(training_op)
        #done training, get final beta:
        best_beta = beta.eval()

    print(best_beta)
    evaluateBetasOverTest(best_beta, featuresZ_pBias_test, price_test)
```

Assumptions:

1. Data is already setup and testL2Reg is called.
2. Momentum is set to 0.9 by default. User can change the value by passing new value as parameter during function call.

Part II, Question 3

Sub-Question (a)

Given: Jaccard similarity of two sets is 0
To Prove: the estimate one would get from minhashing always yields

<u>Proof</u>:

When we compare 2 columns of a characteristic matrix, we can have the rows of the following type:

1. **Row X:** Has value 1 in both sets
2. **Row Y:** Has value 1 in one of the set and 0 in the other
3. **Row Z:** Has value 0 in both sets

Since row Z is has elements not present in both sets, it is considered for Jaccard Similarity or Minhashing. Hence, we ignore rows of type Z.

Assume,
Number of rows of type X = x
Number of rows of type Y = y

Intersection of the two sets = x … (Since row X has value 1 in both sets)
Union of the two sets = x+y … (Since it contains all rows that have value 1 in at least 1 set)

By definition of Jaccard Similarity,
Jaccard Similarity = $\frac{\textit{Intersection of the two sets}}{\textit{Union of the two sets}} = \frac{x}{x+y}$
Given that, Jaccard Similarity = 0
This is only possible if x = 0 i.e. there is no row which has value 1 in both sets.

By definition of Minhashing,
h(Set) = Number of the first row which has value 1.
The similarity estimation that one gets from Minhashing 2 sets = $\frac{\textit{Number of times both hash values match}}{\textit{Number of permutations}}$

Since, x = 0 (i.e. there is no intersection),
Both sets will have hash as a row number with row type = Y
Since row type Y, has only 1 set with value 1, hash values of both sets will always be different.

Therefore, hash values of both sets will never match.
Hence, Similarity Estimation by Minhashing = $\frac{\textit{Number of times both hash values match}}{\textit{Number of permutations}} = \frac{0}{n} = 0$

Therefore, if the Jaccard similarity of two sets is 0 then the estimate one would get from Minhashing always yields 0.

Part II, Question 3

Sub-Question (b)

```
# Define Globally

# Final Signature Matrix
sig_matrix = dict()
# hash_func is a list of 500 hash functions generated randomly
hash_func = [getHashfunc(i) for i in rand(1, num=500)]

# Map every element to list of all sets it is present in
def map(k):
   pairs = []
   set_name = k[0]
   element_list = k[1]

   for e in element_list:
      pairs.append(e, [set_name])
   return pairs

# Concatenate list of all sets for a particular key element
def reduceByKey(a, b):
   return (a + b)

# Run minhashing algorithm for each element
def reduce(vs):
   element = vs[0]
   set_list = vs[1]

   # Pre-compute all hash values for this element
   h = []
   for i in range(500):
      h.append(hashes[i](element))

   # Traverse through every set in the list
   for s in set_list:

      # Update every row of final signature matrix of column = set
      for i in range(500):

         # Get ith hash value
         hash_val = h[i]

         try:
            # Find current value of ith row in sig_matrix
            curr_sig_val = sig_matrix[s][i]

         except KeyError:
            # If set not present in sig_matrix dictionary, all values of set = inf (Not updated yet)
            sig_matrix[s] = [float('-inf')] * 500
            curr_sig_val = float('inf')
```

```
        # Update sig_matrix row with minimum of current value and hash value
        if hash_val < curr_sig_val:
            sig_matrix[s][i] = hash_val
```

 # Function for each RDD of the form: (set_name, list_of_one_or_more_elements)
def Minhashing_Spark(rdd):

```
    pairs_after_map = rdd.flatMap(lambda x: map(x))
    pairs_reduced_by_key = pairs_after_map.reduceByKey(lambda x: reduceByKey(x))
    pairs_reduced_by_key.foreach(lambda x: reduce(x))
```

<u>Assumptions</u>:

1. Signature Matrix is defined globally and can be accessed by all RDD's.
2. Hash functions can be accessed globally and 500 hash functions are randomly generated.

<u>Description of Spark Transformations</u>:

1. **FlatMap**
   This maps each rdd of the form (set_name, list_of_one_or_more_elements)
   Key value pair generated for each element in the list is: (element, [set_name])

2. **ReduceByKey**
   Reduces all (Key, [Value]) pairs having same Key to (Key, [$Value_1$, $Value_2$, …, $Value_N$]).
   All sets for a particular element are concatenated.

3. **For Each**
   For each RDD of the form (element, list_of_sets), run the reduce function.
   The reduce function has the minhashing algorithm.
   Here it updates the globally declared signature matrix for each element.