Problem Set 1

Exercise 1

1. a).

$1 - (1-p)^n$. We know that $P_0 = (1-p)^n$, therefore the probability of no machines fails during the time is:

$$1 - P_0 = 1 - (1-p)^n$$

1. b).

$$P_k = C_n^k p^k (1-p)^{n-k}$$

1. c).

We know that $P_0 + P_1 + \cdots + P_n = 1$, therefore $P_1 + \cdots + P_n = 1 - P_0 = 1 - (1-p)^n$, we can also demonstrate the result by calculating the sum directly. We know

$$P_k = C_n^k p^k (1-p)^{n-k}$$

Then $P_1 + P_2 + \cdots + P_n = \sum_{k=1}^{n} C_n^k p^k (1-p)^{n-k} = \sum_{k=1}^{n} C_n^k p^k (1-p)^n (1-p)^{-k}$

$$= (1-p)^n \sum_{k=1}^{n} C_n^k \left(\frac{p}{1-p}\right)^k = (1-p)^n \left[\sum_{k=0}^{n} C_n^k \left(\frac{p}{1-p}\right)^k - C_n^0 \left(\frac{p}{1-p}\right)^0\right]$$

$$= (1-p)^n \left[\left(1 + \frac{p}{1-p}\right)^n - 1\right] = 1 - (1-p)^n$$

Exercise 2:

2. a)

The descriptions usually come from the spark official website: spark.apache.org. The description of the transformations of the join(), sort() (sortby or sortByKey) and groupby() are as follows:

i. join()

join(otherDataset, [numTasks]): When called on datasets of type (K, V) and (K, W), returns a dataset of (K, (V, W)) pairs with all pairs of elements for each key. Input is pairs of the dataset, output is also pairs of the dataset with the data of the same key value joined together. See example:

Input:

```
stuMark = sc.parallelize([('student1', 7.5), ('student2', 8), ('student3', 9)])
stuGen = sc.parallelize([('student1', 'M'), ('student2', 'F'), ('student3', 'M')])
stdInf = stuMark.join(stuGen).collect()
```

Output:

```
[('student2', (8, 'F')), ('student3', (9, 'M')), ('student1', (7.5, 'M'))]
```

ii. sort()

We did not find sort() on the spark.apache website, assuming here it means sortby() or sortByKey(). The description of sortBy() and sortByKey() is as follows:

- sortByKey(self, ascending=True, numPartitions=None, keyfunc=lambda x: x): Sorts this RDD, which is assumed to consist of (key, value) pairs.

  Input:

  ```
  tmp = [('a', 1), ('b', 2), ('1', 3), ('d', 4), ('2', 5)]
  sortByKeyResult = sc.parallelize(tmp).sortByKey().collect()
  print(sortByKeyResult)
  ```

  Output:

  ```
  [('1', 3), ('2', 5), ('a', 1), ('b', 2), ('d', 4)]
  ```

- sortBy(self, keyfunc, ascending=True, numPartitions=None): Sorts this RDD by the given keyfunc, default is an ascending order

  Input:

  ```
  tmp = [('a', 1), ('b', 2), ('1', 3), ('d', 4), ('2', 5)]
  sortedBySecondTerm = sc.parallelize(tmp).sortBy(lambda x: x[1]).collect()
  print (sortedBySecondTerm)
  ```

  Output:

  ```
  [('a', 1), ('b', 2), ('1', 3), ('d', 4), ('2', 5)]
  ```

iii. groupBy()

groupBy(self, f, numPartitions=None), Return an RDD of grouped items, here f is a function handler.

Input:

```
groupRDD = sc.parallelize([1, 1, 2, 3, 5, 8])
groupResult = groupRDD.groupBy(lambda x: x % 2).collect()
print(groupResult)
sortGroupResult = sorted([(x, sorted(y)) for (x, y) in groupResult])
print(sortGroupResult)
```

Output:

```
[(0, <pyspark.resultiterable.ResultIterable object at 0x7f83754a7eb8>), (1, <pyspark.resultit
erable.ResultIterable object at 0x7f83754b7e48>)]
[(0, [2, 8]), (1, [1, 1, 3, 5])]
```

2. b)

Three transformation examples are given: flatmap, groupby, sortByKey

The source code is as follows:

```
import numpy as np
import sys
from random import random
from pyspark import SparkContext

if __name__ == "__main__":
    sc = SparkContext(appName = 'TransformationsTest')
    iniRDD = sc.parallelize([1,2,3])

    # test the transformation: flatmap
    flatMapRDD = iniRDD.flatMap(lambda x: (x, x+x, x**x))

    # test the transformation: groupBy
    GroupByRDD = flatMapRDD.groupBy(lambda x: x)

    # test the transformation: sortByKey
    sortGroupRDD = GroupByRDD.sortByKey()

    # prepare the result for output
    grouplist = GroupByRDD.map(lambda x: (x[0], list(x[1])))
    sortGrouplist = sortGroupRDD.map(lambda x: (x[0], list(x[1])))

    print(iniRDD.collect())
    print(flatMapRDD.collect())
    # count how many groups have been generated
    print(GroupByRDD.count())
    print(grouplist.collect())
    print(sortGrouplist.collect())
```

The result is as follows:

```
[1, 2, 3]
[1, 2, 1, 2, 4, 4, 3, 6, 27]
6
[(4, [4, 4]), (1, [1, 1]), (2, [2, 2]), (6, [6]), (3, [3]), (27, [27])]
[(1, [1, 1]), (2, [2, 2]), (3, [3]), (4, [4, 4]), (6, [6]), (27, [27])]
```

Exercise 3:

3. a)

The description of each step in the for-loop is given out in the comment part of the code. The detailed explanation could be referred in the source code.

```
while tempDist > convergeDist:

    # data.map here returns a  tuple of three dimension
    # [the index of where the point belongs to, point values, 1]
    # the index will be used as the key to differntiate which cluster the points belongs to

    closest = data.map(
        lambda p: (closestPoint(p, kPoints), (p, 1)))
    closestlist = closest.map(lambda x: (x[0], list(x[1])))

    # For the points with the same index, the value will be group togetherr
    # here pl_cl[0] is the point (location) value, pl_cl[1] equals to one, and will be used
    # as tracking how many points in this cluster
    pointStats = closest.reduceByKey(
        lambda p1_c1, p2_c2: (p1_c1[0] + p2_c2[0], p1_c1[1] + p2_c2[1]))

    # this returns the new clustroid in each cluster, st[0] is the cluster id,
    # st[1][0] is the sume of point location values, st[1][1] is the number of points in the cluster
    newPoints = pointStats.map(
        lambda st: (st[0], st[1][0] / st[1][1])).collect()

    # the tempDist is calculated for testing convergence when it is compared to
    # convergeDist in the while loop. The tempDist is caculated as the sum of the
    # distance between each point in the cluster and the centroid.
    tempDist = sum(np.sum((kPoints[iK] - p) ** 2) for (iK, p) in newPoints)

    # update the centroid of each cluster into variable kPoints for the next iteration
    print ('newPoints: ', newPoints)
    for (iK, p) in newPoints:
        kPoints[iK] = p
```

3. b)

Please refer to the source code to see how GroupByKey is implemented. The scalability of the solution by GroupByKey is bad. That is because when calling GroupByKey, all the key-value pairs are shuffled around, and a lot of unnecessary data is being transferred over the network. When the data is large, and using GroupByKey may load the whole array of values with the same key into memory and it may cause a lot of memory issue.

(for reference:)
https://databricks.gitbooks.io/databricks-spark-knowledge-base/content/best_practices/prefer_reducebykey_over_groupbykey.html

Exercise 4:

a).

Broadcast variables allow the programmer to keep a read-only variable cached on each machine rather than shipping a copy of it with tasks. They can be used, for example, to give every node a copy of a large input dataset in an efficient manner. Spark also attempts to distribute broadcast variables using efficient broadcast algorithms to reduce communication cost.

After the broadcast variable is created, it should be used instead of the value v in any functions run on the cluster so that v is not shipped to the nodes more than once. In addition, the object v should not be modified after it is broadcast in order to ensure that all nodes get the same value of the broadcast variable (e.g. if the variable is shipped to a new node later).

(refer to http://spark.apache.org)

b).

Exercise 5:

The code is as follows:

```
128
129     centroids = data.takeSample(False, K, 1)
130     newCentroids = centroids[:]  # creates a copy
131
132     # Visualization of the Clusters
133     plt.ion()
134     plt.figure(figsize=(8, 6))
135
136     counter = 0
137     tempDist = 2 * convergeDist
138     while tempDist > convergeDist:
139         print("\n### Iteration#: %d" % (counter))
140         counter += 1
141
142         closest = data.map(lambda p: (closestPoint(p, centroids), (p, 1)))
143
144         visualizeClusters(closest.collect(), centroids, counter)
145
146         for cIndex in range(K):
147             closestOneCluster = closest.filter(lambda d: d[0] == cIndex).map(lambda d: d[1])
148             print("Cluster with index %d has %d points" % (cIndex, closestOneCluster.count()))
149             sumAndCountOneCluster = closestOneCluster.reduce(lambda p1, p2: (p1[0] + p2[0], p1[1] + p2[1]))
150
151             vectorSum = sumAndCountOneCluster[0]
152             count = sumAndCountOneCluster[1]
153             newCentroids[cIndex] = vectorSum / count
154
155         tempDist = distanceCentroidsMoved(centroids, newCentroids)
156         print("*tempDist=%f\n*centroids=%s\n*newCentroids=%s" % (tempDist, str(centroids), str(newCentroids)))
157         centroids = newCentroids[:]  # creates a copy
158
159     print("\n=== Final centers: " + str(centroids))
160
```

There are three cases:

Case 1: on line 130: newCentroids = centroids[:] , and line 157: centroids = newCentroids[:]
Case2: on line 130: newCentroids = centroids[:], and line 157: centroids = newCentroids
Case3: on line 130: newCentroids = centroids, and line 157: centroids = newCentroids[:]
Case4: on line 130: newCentroids = centroids, and line 157: centroids = newCentroids

In Case 1, the original case: in line 130, the initial centroids value is copied to newCentroids, in the for loop, newCentroids is updated and therefore the following tempDist is updated, and in line 157, the updated newCentroids value is copied back to centroids.

In Case 2, in line 130, the initial centroids value is copied to newCentroids. In the first and second for loop, newCentroids is updated. However, in line 157, the reference of newCentroids is copied to centroids, which will refer to the same list as newCentroids refer to. This means in the second iteration, in line 155 when calculating the tempDist, both newCentroids and centroids refer to the same list, and tempDist will be zero and the while loop will be stopped.

Thus, the result will be different, the iteration will be implemented only twice. Also the result of centroids is also only updated twice.
In Case 3, in line 130 the list which centroid refers to is copied so that newCentroids will refer to the same list. In the first iteration, newCentroids is updated, which means the value in the list which centroid refers to is also updated. The tempDist will have value of zero when it is calculated for the first time. In line 157, the list value which newCentroids is refered to will be copied to centroids, now newCentroids and centroids refers to different list and have the same value. But the while loop is stopped because of the tempDist is equal to zero after the first iteration. Case 1, Case 2 and Case 3 will have different result. In Case 3, the iteration is only implemented once, and also the result of centroids is updated once.

In Case 4, the result will be same as Case three until line 157. While in 157, the reference of newCentroids is copied back to centroids, which means they refer to the same list. As to the result, Case 4 will have the same result as Case 3, i.e. the iteration is implemented only once and centroids is updated once.