Lab 7

Group 2: Yiding Chang, Kexuan Huang, Yifan Shen, Qinghang Wu

Pre-processing

Import the csv file PBMC_16k_RNA.csv into a spark dataframe.

```
from pyspark import SparkContext
from pyspark.sql import SQLContext

sc= SparkContext()
sqlContext = SQLContext(sc)

spark_df = sqlContext.read.options(header='true',
inferSchema='true').csv('PBMC/PBMC_16k_RNA.csv')
```

Find the range of data in the first two features KLHL17 and HES4 inside the dataframe.

```
spark_df.agg({'KLHL17': 'max'}).show()
spark_df.agg({'KLHL17': 'min'}).show()

+-----------+
|max(KLHL17)|
+-----------+
|       10.0|
+-----------+

+-----------+
|min(KLHL17)|
+-----------+
|-0.11507928|
+-----------+

spark_df.agg({'HES4': 'max'}).show()
spark_df.agg({'HES4': 'min'}).show()

+---------+
|max(HES4)|
+---------+
| 9.472544|
+---------+

+---------+
|min(HES4)|
+---------+
|-1.339669|
+---------+
```

Therefore, the range of KLHL17 is from -0.1151 to 10.0 and the range of HES4 is from -1.3397 to 9.4725.

Is it necessary to perform data standardization? Explain.

As the range of KLHL17 and HES4 are approximately the same, we can assume it is not necessary to perform data standardization. However, we can still standardize the data if we want.

PCA Analysis

Why could PCA be useful to analyse this dataset?

PCA is mainly used to reduce dimensions and find the most important features while neglecting the less important (trivial) ones. As our dataset has 1882 features, PCA becomes useful for us to reduce the number of features and recognize the most important ones. In this way, it will be much easier for fruther analysis.

Use org.apache.spark.mllib.feature.PCA to perform PCA analysis on the dataset. For now only retrieve the first two principal components.

We first reformat the column header because we can not regonize the column headers in the format like 'xxxx.x'. Then, we use the `VectorAssembler` from the same library to prepare for PCA analysis.

```
pca = PCA(k=2, inputCol='features', outputCol='pca_features')
pca_model = pca.fit(features)
result = pca_model.transform(features).select("pca_features")
result.show(2)
```

```
+--------------------+
|        pca_features|
+--------------------+
|[7.46652731979110...|
|[-4.4924025163180...|
+--------------------+
only showing top 2 rows
```

How much of the dataset do the two first principal components explain?

We use the following code,

```
print(pca_model.explainedVariance.toArray())

[0.02942326 0.01307317]
```

and find out that the first rwo principal components explain the features 0.02942, 0.01307.

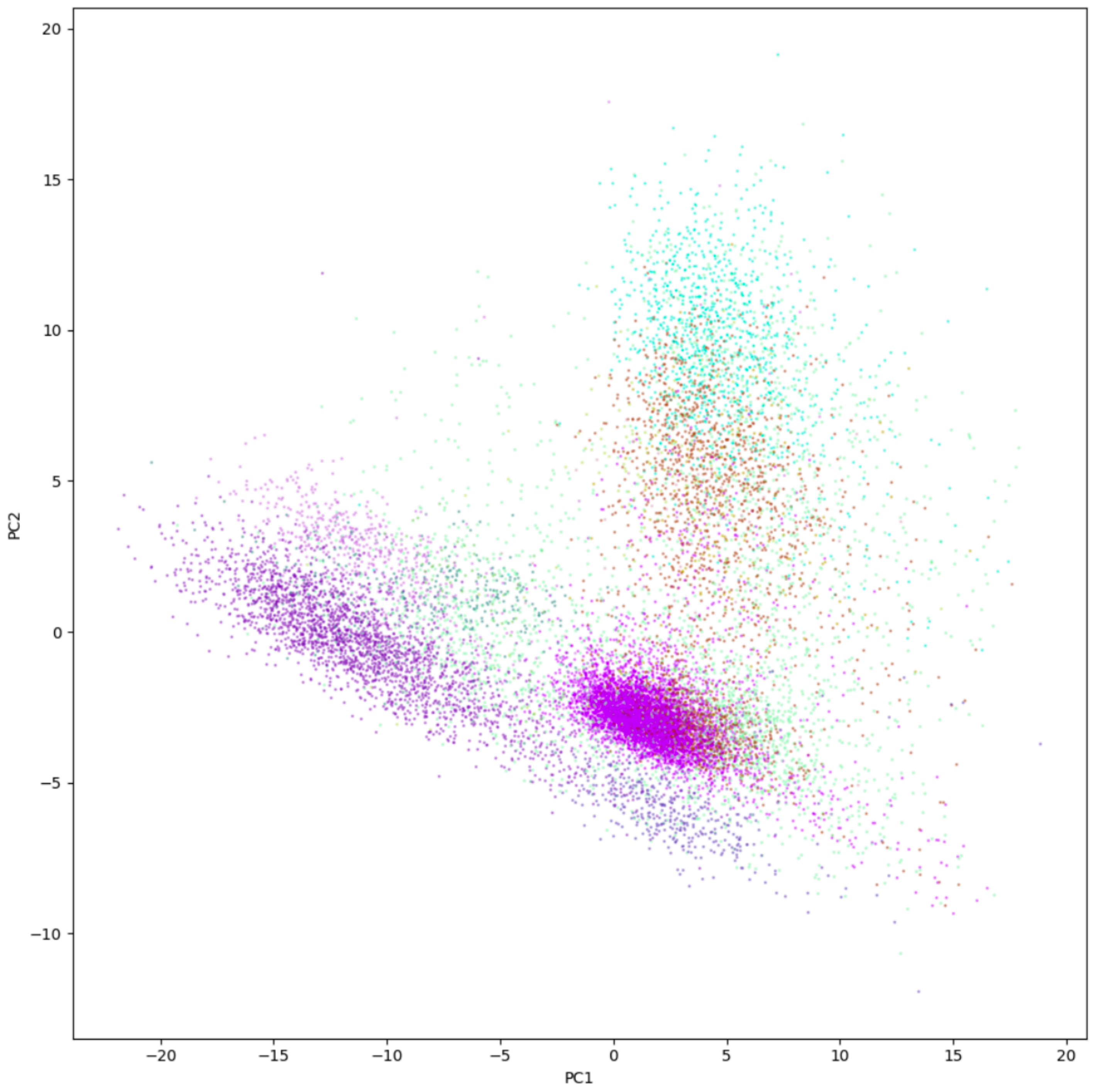
In this context how useful is PCA?

This is relatively small numbers considering they are the two most important features. However, as there are in total 1882 features, it still makes sense when two of the most important features can explain 4.2% of the variance. So the PCA is still useful, but not much useful.

Import the csv file PBMC_16k_RNA_label.csv, which contains the type of cell for each index.

```
label = sqlContext.read.options(header='true',
inferSchema='true').csv('PBMC/PBMC_16k_RNA_label.csv')
```

Plot a scatter plot with the obtained two principal components, with different types of cell in different colours.

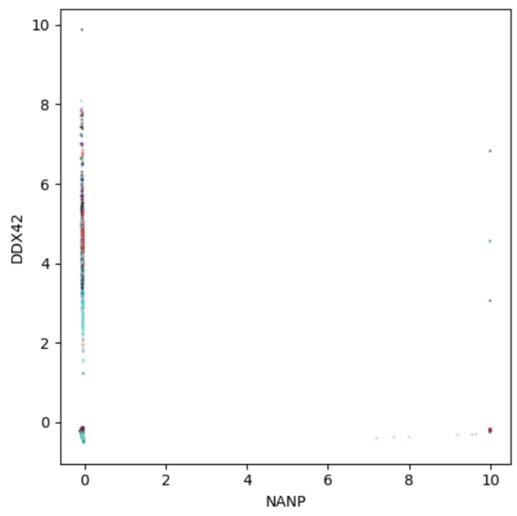


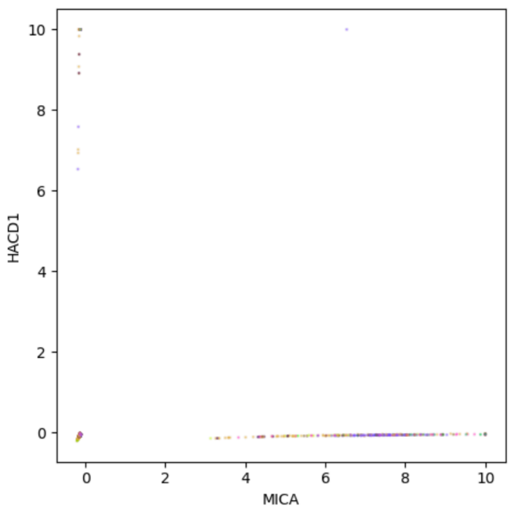
What can you observe?

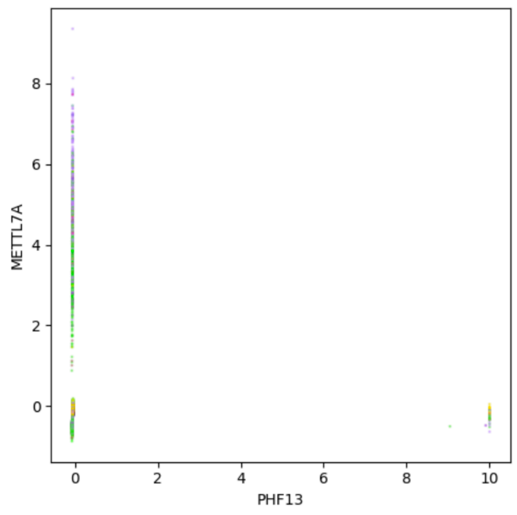
- The same kind of cells (data points labeled with the same color) usually cluster together on the plot
- The two principal components can fairly distinugish different kinds of cells.

Plot another plot using two random columns of the original dataframe.

I ran a series of plots with two random columns. Some of the result is as follows,







Describe the difference in the two plots.

For the plot using two random columns, there is no obvious patterns about different kinds of cells. All kinds of cells are located close to each other. In other words, the plot with two principal components can distinguish the clusters, but the one with two random columns can not (there is no distinguishable patterns). Therefore, the same kind of cells will have similar values regarding the principal components.

Categorization using Gradient Descent

Prepare two subsets of the datasets: 70% as training set and 30% as test set. Frank reminds you to only use the training set to the perform logistic regression. Determine the type of cell with the most data points and call it A cells, labelled (1) while non-A cells are labelled (0).

We use `feature_df.groupBy('CITEsort').count().show()` to determine the type of cell with the most data points.

```
+--------+-----+
|   Cell|count|
+-------+-----+
| C-mono| 2313|
|    mDC|  303|
| CD8+ T| 2035|
| B cell|  414|
|    ACT| 2952|
|    iNK|  113|
|CD4+ DC|  166|
|    mNK| 1057|
|    DNT|  178|
| CD4+ T| 5262|
|CD8+ DC|   81|
|NC-mono|  537|
+-------+-----+
```

The result we find out is `CD4+ T` cell with 5262 occurences. We label the cells accordingly with the `LabeledPoint` from `pyspark.mllib.regression` . A-cells are labeled as 1 and non-A cells are labeled as 0. Then, the training sets and the testing sets are split randomly in the proportion of 70%-30%.

```
(training, testing) = data.randomSplit([0.7, 0.3])
```

To categorize A and non-A cells, we use spark.mllib.classification.LogisticRegressionWithLBFGS with the principal components p1 and p2. Run Logistic Regression with gradient descent on the training dataset, and apply your model to predict the label in the test set.

Note that `pyspark.ml.linalg.Vector` and `pyspark.mllib.linalg.Vector` may not be compatible, so we need to use the following code to transform `pyspark.ml` dataframe to `pyspark.mllib` dataframe.

```
from pyspark.mllib.util import MLUtils
df = MLUtils.convertVectorColumnsFromML(df, "features")
```

We run the regression with the principal components p1 and p2 with the following command,

```
from pyspark.mllib.classification import LogisticRegressionWithLBFGS
lm = LogisticRegressionWithLBFGS.train(training, iterations=15)
```

The model is then applied to predict the label on the testing set,

```
labels_and_predictions = test_data.map(lambda x: (x.label, model.predict(x.features)))
error_rate = labels_and_predictions.filter(lambda res: res[0] != res[1]).count() /
float(test_data.count())
```

With both the predicted label and the true label, we calculate the error rate. The calculation result is 0.258 for 10 iterations and 0.254 for 15 iterations,

What can you report to Reapor, and more importantly should you be afraid of his reaction?

Maybe not as the error rate is very high. I am afraid he will not be satisfactory with the current result. With such high error rate, we might not categorize the cells very well. One of the underlying reason may be that we use only 2 principal components.

Propose ways to improve the model.

- Introduce a few more principal components
- Use other solvers