

```
[18]: # Environment setup successful loading here should mean all dependencies correctly installed #
import numpy as np
import scanpy as sc
import matplotlib.pyplot as plt
import seaborn as sns
from scipy import stats
import matplotlib.pyplot as plt
```

Load in the dataset

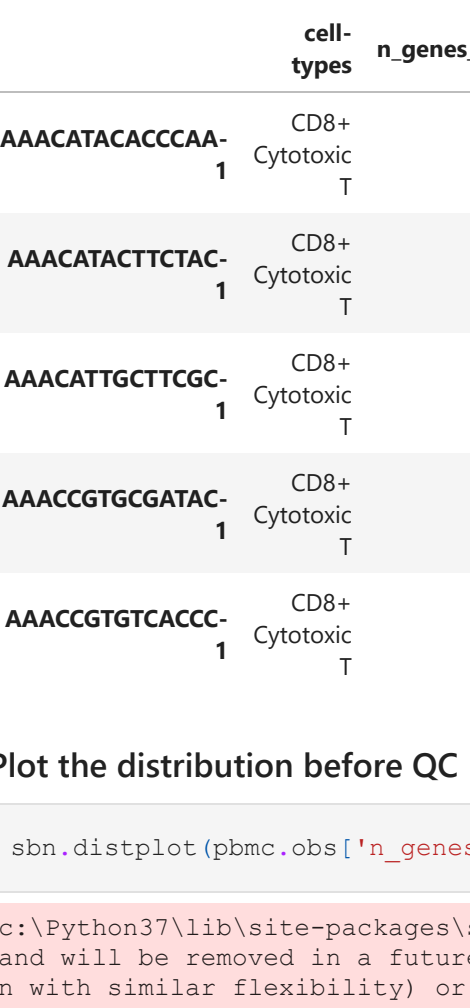
```
In [19]: pbmc = sc.read_h5ad('pbmc.h5ad')
print(f'The number of cells are: {pbmc.shape[0]}')
print(f'The number of genes are: {pbmc.shape[1]}')
```

The number of cells are: 22475
The number of genes are: 32738

Check coverage

```
In [20]: plt.spy(pbmc.X[100, :100], markersize=1)
```

```
Out[20]: <matplotlib.lines.Line2D at 0x1f321e49488>
```



QC Metrics

```
In [21]: # inplace saves all of the metrics to pbmc.obs
sc.pp.calculate_qc_metrics(pbmc, inplace=True)
```

```
In [22]: pbmc.obs[0:5]
```

```
Out[22]:
```

	cell- types	n_genes_by_counts	log1p_n_genes_by_counts	total_counts	log1p_total_counts	pct_counts_in_top_50_genes	pct...
AAACATACACCCCA- 1	CD8+ Cytotoxic T	498		6212606	1216.0	7.104144	44.572368
AAACATACTCTAC- 1	CD8+ Cytotoxic T	925		6.830874	2683.0	7.895063	39.619829
AAACATGCTCTGC- 1	CD8+ Cytotoxic T	554		6.318968	1310.0	7.178545	43.358779
AAACCGTGGGATAC- 1	CD8+ Cytotoxic T	687		6.533789	1854.0	7.525640	44.983819
AAACCGTGTACACC- 1	CD8+ Cytotoxic T	471		6.156979	1274.0	7.150702	49.450549

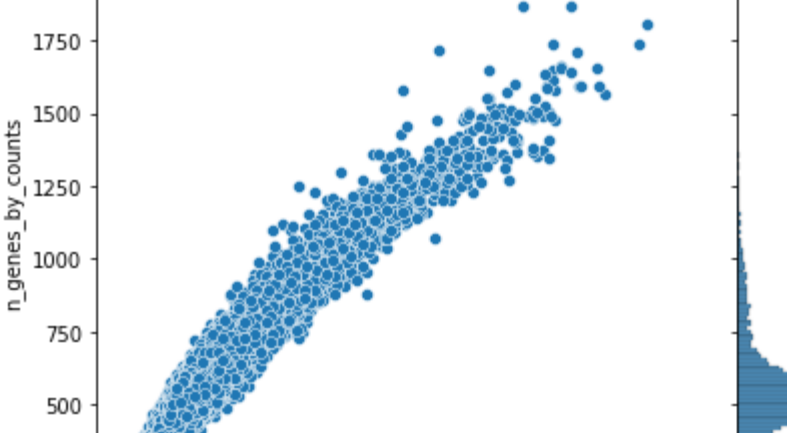
Plot the distribution before QC

```
In [23]: sbn.distplot(pbmc.obs['n_genes_by_counts']).set(title='Distribution before QC')
```

ci\Python37\lib\site-packages\seaborn\distributions.py:2557: FutureWarning: 'distplot' is a deprecated function and will be removed in a future version. Please adapt your code to use either 'displot' (a figure-level function with similar flexibility) or 'histplot' (an axes-level function for histograms).

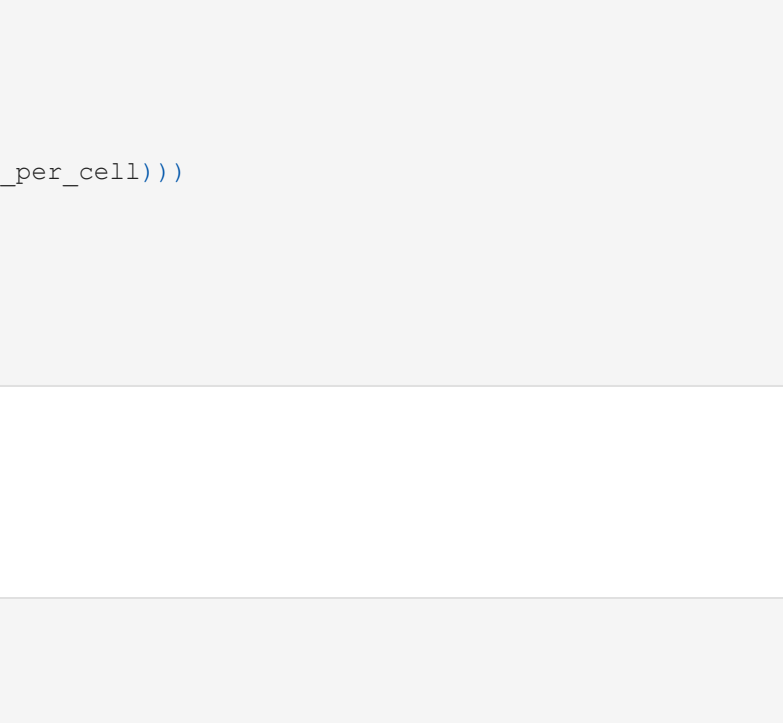
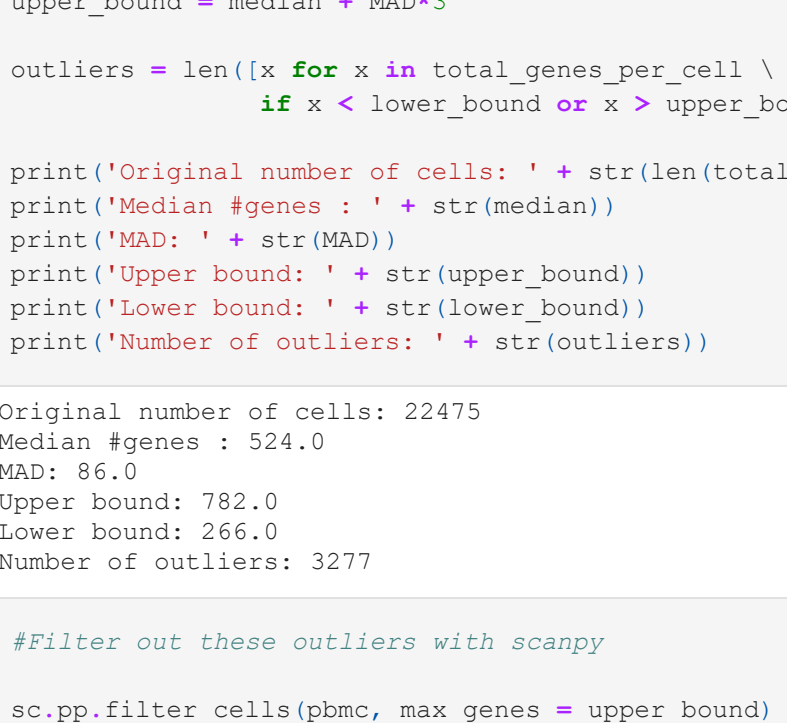
Warnings:Warn(msg, FutureWarning)

```
Out[23]: Text(0.5, 1.0, 'Distribution before QC')
```



Identify cell types

```
In [24]: sc.pl.violin(pbmc, ['n_genes_by_counts', 'total_counts'],
                  jitter=0.4, groupby='cell-types', rotation=20)
```



```
In [25]: p = sbn.countplot(data=pbmc.obs, y='cell-types')
plt.grid()
plt.show()
```

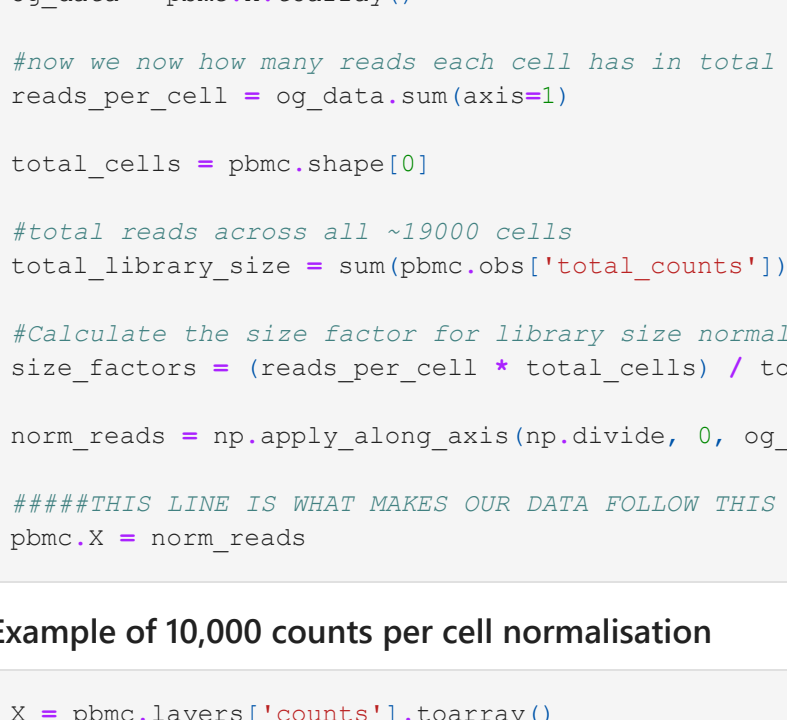


Gene coverage as read counts increase

This plot gives us an indication of how well we're sampling genes clearly the number of genes is underrepresented but the data is beginning to plateau as total read counts increases which is a good sign

```
In [26]: sbn.jointplot(data=pbmc.obs, x='total_counts', y='n_genes_by_counts')
```

```
Out[26]: <seaborn.axisgrid.JointGrid at 0x1f322389c48>
```



Removing cells with too many or too few genes

Using a metric like Median absolute deviation we can remove outliers that are expressing too many or too few genes per cell

```
In [27]: total_genes_per_cell = pbmc.obs['n_genes_by_counts']
median = np.median(total_genes_per_cell)

abs_deviation = abs(total_genes_per_cell - median)
MAD = np.median(abs_deviation)

lower_bound = median - MAD*3
upper_bound = median + MAD*3

outliers = len([x for x in total_genes_per_cell \
                 if x < lower_bound or x > upper_bound])

print('Original number of cells: ' + str(len(total_genes_per_cell)))
print('Median genes: ' + str(median))
print('MAD: ' + str(MAD))
print('Upper bound: ' + str(upper_bound))
print('Lower bound: ' + str(lower_bound))
print('Number of outliers: ' + str(outliers))
```

Original number of cells: 22475
Median genes: 524.0
MAD: 85.0
Upper bound: 782.0
Lower bound: 266.0
Number of outliers: 3277

```
In [28]: # Filter out these outliers with scanpy
sc.pp.filter_cells(pbmc, max_genes = upper_bound)
sc.pp.filter_cells(pbmc, min_genes = lower_bound)

print(f'Number of cells after filtering: {pbmc.shape[0]}')
```

Number of cells after filtering: 19198

Remove genes that are not present in enough cells

We don't want to inflate our data with genes that either don't express at all or aren't in at least 3 cells

```
In [29]: print('Genes before filtering: ', pbmc.shape[1])

sc.pp.filter_genes(pbmc, min_cells=3)

print('Genes after filtering: ', pbmc.shape[1])
```

Genes before filtering: 32738
Genes after filtering: 14945

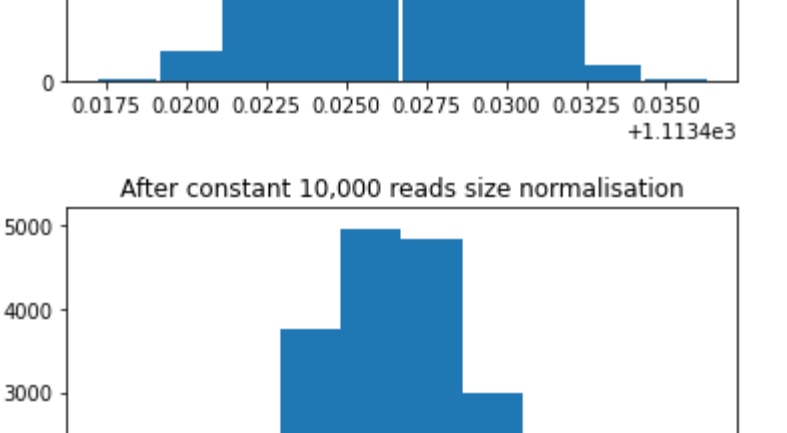
Visualise the changes

```
In [30]: sbn.distplot(pbmc.obs['n_genes_by_counts']).set(title='Distribution after QC')
```

ci\Python37\lib\site-packages\seaborn\distributions.py:2557: FutureWarning: 'distplot' is a deprecated function and will be removed in a future version. Please adapt your code to use either 'displot' (a figure-level function with similar flexibility) or 'histplot' (an axes-level function for histograms).

Warnings:Warn(msg, FutureWarning)

```
Out[30]: Text(0.5, 1.0, 'Distribution after QC')
```



```
In [31]: pbmc.layers['counts'] = pbmc.X.copy()
```

There are several ways to normalise scRNAseq - lets take a look at the differences based on their distributions

Perform normalisation using calculated library size

```
In [32]: og_data = pbmc.X.toarray()

# Now we now how many reads each cell has in total
reads_per_cell = og_data.sum(axis=1)

total_cells = pbmc.shape[0]

# Total reads across all ~19000 cells
total_library_size = sum(pbmc.obs['total_counts'])

# Calculate the size factor for library size normalisation
size_factors = (reads_per_cell * total_cells) / total_library_size

norm_reads = np.apply_along_axis(np.divide, 0, og_data, size_factors)

####THIS LINE IS WHAT MAKES OUR DATA FOLLOW THIS NORMALISATION
pbmc.X = norm_reads
```

Example of 10,000 counts per cell normalisation

```
In [33]: X = pbmc.layers['counts'].toarray()

reads_per_cell = X.sum(axis=1)
after = 1e4
scale_factors = reads_per_cell / after # scale factor for each cell

# Calculating 'counts-per-ten-thousand' normalisation between cells.
x_counts_norm = np.apply_along_axis(np.divide, 0, X, scale_factors)

reads_per_cell_norm = x_counts_norm.sum(axis=1)

# For constant library size
pbmc.layers['constant_1e'] = x_counts_norm

stats.describe(reads_per_cell_norm) # confirm that most cells have total reads ~1e4
```

1105.0
DescribeResult(nobs=19198, minmax=(1104.9912, 1105.0095), mean=1105.0, variance=0.0005933718, skewness=0.06664
06436443329, kurtosis=-0.5836937236907622)

Normalisation by median

```
In [34]: X = pbmc.layers['counts'].toarray()

reads_per_cell = X.sum(axis=1)
median = np.median(reads_per_cell)
print(median)
scale_factors = reads_per_cell / median # scale factor for each cell

# Calculating 'counts-per-ten-thousand' normalisation between cells.
x_counts_norm = np.apply_along_axis(np.divide, 0, X, scale_factors)

reads_per_cell_norm = x_counts_norm.sum(axis=1)

# For constant library size
pbmc.layers['median_norm'] = x_counts_norm

stats.describe(reads_per_cell_norm) # confirm that most cells have total reads ~1e4
```

1105.0
DescribeResult(nobs=19198, minmax=(1104.9912, 1105.0095), mean=1105.0, variance=7.908801e-06, skewness=0.067834
06436443329, kurtosis=-0.5836937236907622)

Now we can compare normalisation methods

```
In [35]: plt.hist(pbmc.layers['counts'].sum(axis=1))
plt.title('Before normalisation')
plt.show()

plt.hist(pbmc.X.sum(axis=1))
plt.title('After calculated library size normalisation')
plt.show()

plt.hist(pbmc.layers['constant_1e'].sum(axis=1))
plt.title('After constant 10,000 reads size normalisation')
plt.show()

plt.hist(pbmc.layers['median_norm'].sum(axis=1))
plt.title('After median normalisation')
plt.show()
```

Log-transformation

We do this to reduce difference between genes that are expressing at very different levels so that we can do our feature selection.

```
In [36]: sc.pp.log1p(pbmc)
```

Feature selection

```
In [37]: # All of the variables I selected are just the
# default values from the docs

# Applies an annotation of variability so that we can plot it
sc.pp.highly_variable_genes(pbmc, min_mean=0.0125, max_mean=3, min_disp=0.5)
```

```
In [38]: genes = pbmc.var['highly_variable']

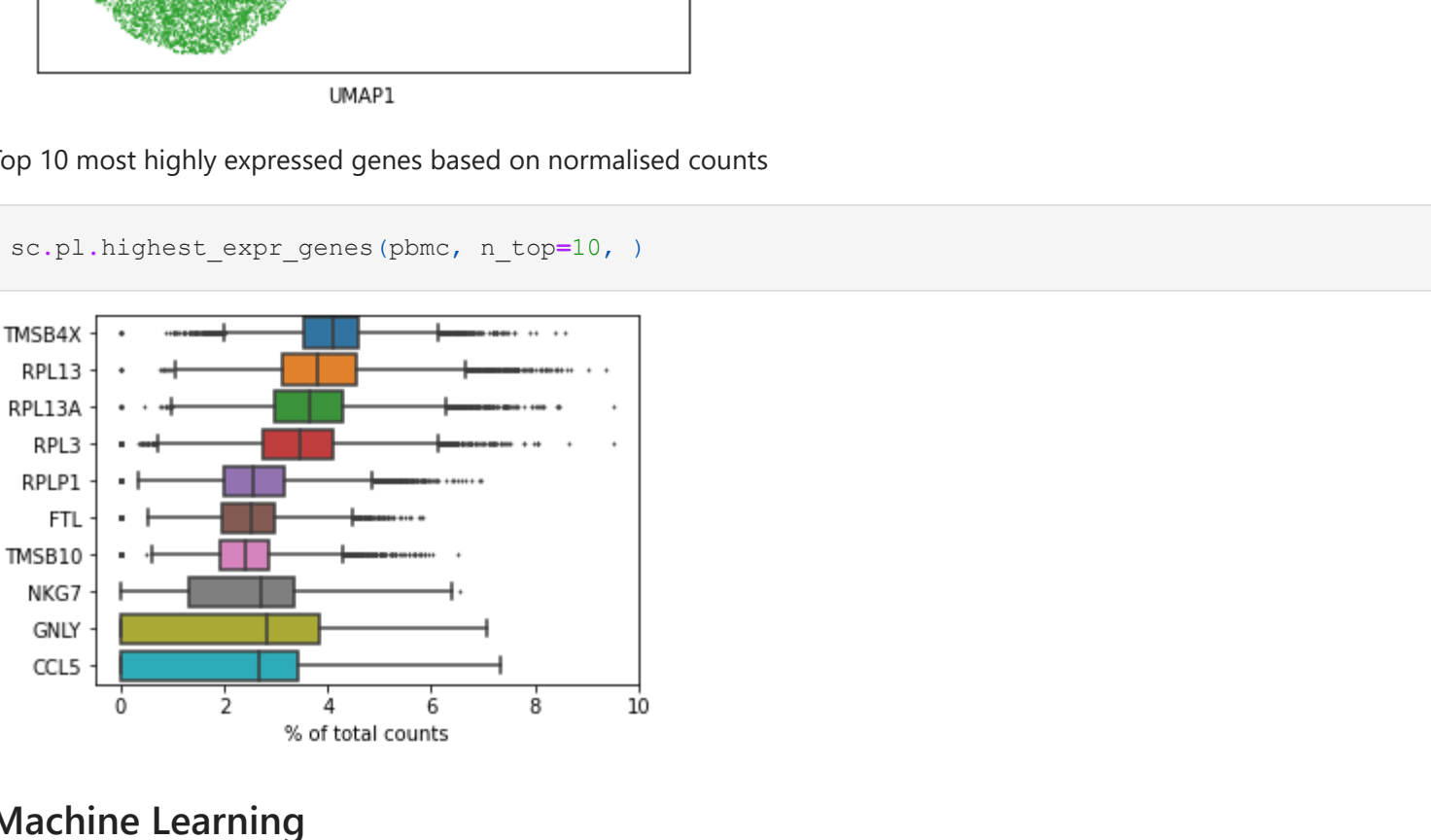
variable_genes = genes[genes == True]

print(f'The number of variable genes are {len(variable_genes)}')
```

The number of variable genes are 903

Some cool plots showing the mean expression and variability of genes

```
In [39]: sc.pl.highly_variable_genes(pbmc)
```



Now we can subset our data set to only highly variable genes

```
In [40]: sc.pp.highly_variable_genes(pbmc, min_mean=0.0125, max_mean=3, min_disp=0.5, subset=True)
```

```
In [41]: print(f'The number of cells are: {pbmc.shape[0]}')
print(f'The number of genes are: {pbmc.shape[1]}')
```

The number of cells are: 19198
The number of genes are: 903

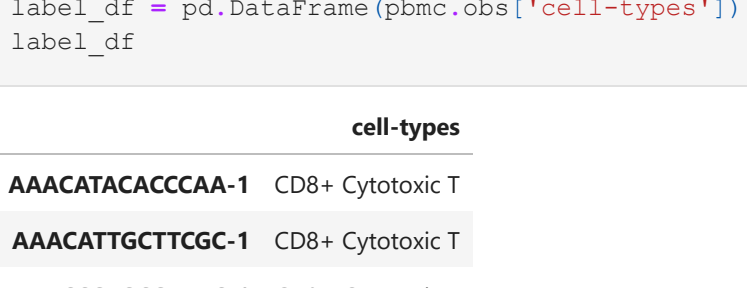
Visualisation

PCA analysis

```
In [42]: sc.tl.pca(pbmc, svd_solver='arpack')
```

```
In [43]: explained_variance = pbmc.uns['pca']['variance_ratio'][:125]

p = sbn.Lineplot(data=explained_variance).set(title='Scree plot', \
                                              xlabel='PC', \
                                              ylabel='Explained variance')
```



```
In [44]: sc.pl.pca(pbmc, components='1,2', annotate_var_explained=True, color='cell-types')
```



UMAP representation

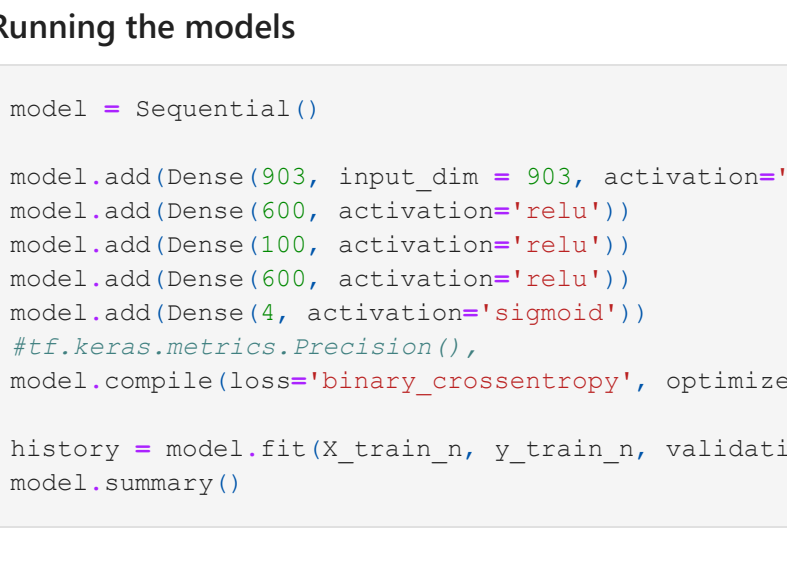
Just experimenting with different cluster sizes here, can check the appropriate number of cluster neighbors

30 Neighbors

```
In [45]: # neighbors is just a function that computes
# pairwise distances between points for UMAP
sc.pp.neighbors(pbmc, n_neighbors=30, n_pcs=40)

# Now with our distances, we can create the UMAP plot
sc.tl.umap(pbmc)

sc.pl.umap(pbmc, color='cell-types')
```




15 Neighbours

```
In [46]: sc.pp.neighbors(pbmc, n_neighbors=15, n_pcs=13)

# Now with our distances, we can create the UMAP plot
sc.tl.umap(pbmc)

sc.pl.umap(pbmc, color='cell-types')
```



```
In [47]: sc.pp.neighbors(pbmc, n_neighbors=15, n_pcs=20)

# Now with our distances, we can create the UMAP plot
sc.tl.umap(pbmc)

sc.pl.umap(pbmc, color='cell-types')
```



```
In [77]: # Now lets check the UMAP based on the elbow of the scree plot, which showed around PC4 = 4

sc.pp.neighbors(pbmc, n_neighbors=15, n_pcs=4)

sc.tl.umap(pbmc)

sc.pl.umap(pbmc, color='cell-types')
```


Top 10 most highly expressed genes based on normalised counts

```
In [65]: sc.pl.highest_expr_genes(pbmc, n_top=10, )
```


Machine Learning

Using an autoencoder architecture with Stochastic gradient descent optimiser, KNN and random forest classification

```
In [48]: # Imports
import pandas as pd
import tensorflow as tf
from tensorflow import keras
from keras.models import Sequential
from keras.layers import Dense
from keras.wrappers.scikit_learn import KerasClassifier
from keras.utils import np_utils
from sklearn.model_selection import train_test_split
import matplotlib.pyplot as plt
import numpy as np
import seaborn as sns
from sklearn.metrics import confusion_matrix
from sklearn import metrics
from sklearn.preprocessing import LabelEncoder
```

Using TensorFlow backend.

```
In [49]: filit_df = pd.DataFrame(pbmc.X, index = pbmc.obs_names, columns = pbmc.var_names)
filt_df
```

```
Out[49]:
```

	HES4	ISG15	TNFRSF18	TNFRSF4	ACAP3	MRP120	RBP7	PGD	FBXO44	AGTRAP	...	MRPL39	USP16	C2orf1
AAACATACACCCCA- 1	0.0	0.000000	0.0	0.0	0.0	0.650055	0.0	0.000000	0.000000	0.0	...	0.0	0.0	0.0
AAACATGCTCTGC- 1	0.0	0.000000	0.0	0.0	0.0	0.000000	0.0	0.000000	0.000000	0.0	...	0.0	0.0	0.0
AAACCGTGGGATAC- 1	0.0	0.000000	0.0	0.0	0.0	0.470350	0.0	0.000000	0.000000	0.0	...	0.0	0.0	0.0
AAACCGTGTACACC- 1	0.0	1.011345	0.0	0.0	0.0	0.000000	0.0	0.000000	0.628421	0.0	...	0.0	0.0	0.0
AAACGACGGGTGGA- 1	0.0	0.000000	0.0	0.0	0.0	0.821195	0.0	0.000000	0.000000	0.0	...	0.0	0.0	0.0
TTTGACTACTGG- 8
TTTCTACTGGGACA- 8	0.0	0.000000	0.0	0.0	0.0	0.000000	0.0	0.000000	0.000000	0.0	...	0.0	0.0	0.0
TTTGACTGAAGTAG- 8	0.0	0.000000	0.0	0.0	0.0	0.000000	0.0	0.538942	0.000000	0.0	...	0.0	0.0	0.0
TTTGACTGGCCCAA- 8	0.0	0.000000	0.0	0.0	0.0	0.664478	0.0	0.000000	0.000000	0.0	...	0.0	0.0	0.0
TTTGACTGTGCGAA- 8	0.0	0.000000	0.0	0.0	0.0	0.000000	0.0	0.000000	0.000000	0.0	...	0.0	0.0	0.0

19198 rows x 903 columns

```
In [50]: label_df = pd.DataFrame(pbmc.obs['cell-types'])
label_df
```

```
Out[50]:
```

	cell-types
AAACATACACCCCA-1	CD8+ Cytotoxic T
AAACATGCTCTGC-1	CD8+ Cytotoxic T
AAACCGTGGGATAC-1	CD8+ Cytotoxic T
AAACCGTGTACACC-1	Dendritic
AAACGACGGGTGGA-1	Dendritic
...	...
TTTGACTACTGG-8	CD8+ Cytotoxic T
TTTCTACTGGGACA-8	CD8+ Cytotoxic T
TTTGACTGAAGTAG-8	CD8+ Cytotoxic T
TTTGACTGGCCCAA-8	CD8+ Cytotoxic T
TTTGACTGTGCGAA-8	CD8+ Cytotoxic T

19198 rows x 1 columns

One Hot Encoding

```
In [51]: labels = label_df['cell-types'].values
encoder = LabelEncoder()
encoder.fit(labels)

encoded_y = encoder.transform(labels)
encoded_y
```

```
Out[51]: array([2, 2, 2, ..., 2, 2, 2])
```

```
In [52]: # Turns the encodings into vectors
dummy_y = np_utils.to_categorical(encoded_y)

# Let's say we want to split the data in 75 / 12.5 / 12.5 for train / valid / test dataset
X_train_n, X_remain_n, y_train_n, y_remain_n = train_test_split(X_remain_n, y_remain_n, test_size=0.3, shuffle=True, random_state=42)

# Now since we want the valid and test size to be equal (10% each of overall data).
# we have to define valid_size=0.5 (that is 50% of remaining data)
X_val_n, X_test_n, y_val_n, y_test_n = train_test_split(X_remain_n, y_remain_n, test_size=0.3, shuffle=True, random_state=42)

print(X_train_n.shape, print(y_train_n.shape)
print(X_test_n.shape), print(y_test_n.shape)
```

```
Out[52]: (None, None)
```

Running the models

```
In [53]: model = Sequential()

model.add(Dense(903, input_dim = 903, activation='relu'))
model.add(Dense(600, activation='relu'))
model.add(Dense(100, activation='relu'))
model.add(Dense(600, activation='relu'))
model.add(Dense(4, activation='sigmoid'))

fit_keras = model.compile(loss='binary_crossentropy', optimizer='SGD', metrics=['accuracy', tf.keras.metrics.Recall()])

history = model.fit(X_train_n, y_train_n, validation_data = (X_val_n, y_val_n), epochs = 30, batch_size = 24)

model.summary()
```



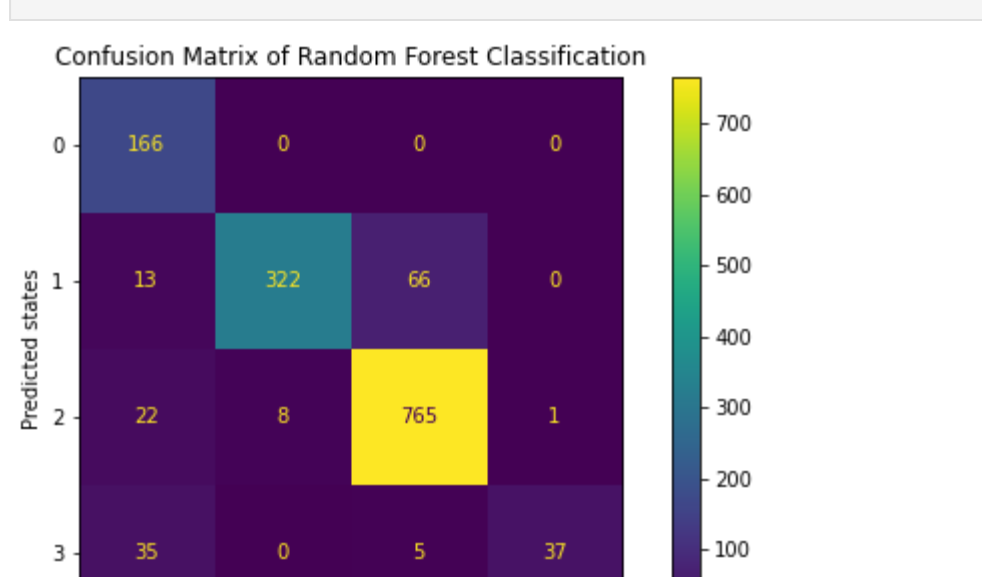
```
Train on 14398 samples, validate on 3360 samples
Epoch 3/30: 14398/14398 [=====] - 8s 565us/step - loss: 0.3849 - accuracy: 0.8316 - recall: 0.8181
- val_loss: 0.2257 - val_accuracy: 0.9240 - val_recall: 0.6492
14398/14398 [=====] - 9s 617us/step - loss: 0.1692 - accuracy: 0.9354 - recall: 0.7155
- val_loss: 0.1533 - val_accuracy: 0.9339 - val_recall: 0.7562
Epoch 4/30: 14398/14398 [=====] - 8s 528us/step - loss: 0.1358 - accuracy: 0.9429 - recall: 0.7791
- val_loss: 0.1361 - val_accuracy: 0.9391 - val_recall: 0.7957
Epoch 5/30: 14398/14398 [=====] - 8s 566us/step - loss: 0.1034 - accuracy: 0.9573 - recall: 0.8433
- val_loss: 0.1138 - val_accuracy: 0.9522 - val_recall: 0.8367
Epoch 6/30: 14398/14398 [=====] - 8s 510us/step - loss: 0.0973 - accuracy: 0.9598 - recall: 0.8536
- val_loss: 0.1100 - val_accuracy: 0.9513 - val_recall: 0.8580
Epoch 7/30: 14398/14398 [=====] - 7s 499us/step - loss: 0.0944 - accuracy: 0.9607 - recall: 0.8617
- val_loss: 0.1078 - val_accuracy: 0.9538 - val_recall: 0.8653
Epoch 8/30: 14398/14398 [=====] - 7s 518us/step - loss: 0.0904 - accuracy: 0.9622 - recall: 0.8686
- val_loss: 0.0987 - val_accuracy: 0.9574 - val_recall: 0.8714
Epoch 9/30: 14398/14398 [=====] - 7s 489us/step - loss: 0.0880 - accuracy: 0.9635 - recall: 0.8741
- val_loss: 0.1012 - val_accuracy: 0.9574 - val_recall: 0.8765
Epoch 10/30: 14398/14398 [=====] - 7s 507us/step - loss: 0.0850 - accuracy: 0.9641 - recall: 0.8788
- val_loss: 0.1312 - val_accuracy: 0.9405 - val_recall: 0.8805
Epoch 11/30: 14398/14398 [=====] - 7s 515us/step - loss: 0.0828 - accuracy: 0.9650 - recall: 0.8821
- val_loss: 0.0964 - val_accuracy: 0.9591 - val_recall: 0.8839
Epoch 12/30: 14398/14398 [=====] - 7s 511us/step - loss: 0.0804 - accuracy: 0.9662 - recall: 0.8857
- val_loss: 0.0957 - val_accuracy: 0.9605 - val_recall: 0.8873
Epoch 13/30: 14398/14398 [=====] - 7s 516us/step - loss: 0.0783 - accuracy: 0.9671 - recall: 0.8889
- val_loss: 0.1068 - val_accuracy: 0.9533 - val_recall: 0.8903
Epoch 14/30: 14398/14398 [=====] - 7s 508us/step - loss: 0.0767 - accuracy: 0.9679 - recall: 0.8916
- val_loss: 0.1039 - val_accuracy: 0.9550 - val_recall: 0.8929
Epoch 15/30: 14398/14398 [=====] - 8s 529us/step - loss: 0.0755 - accuracy: 0.9687 - recall: 0.8942
- val_loss: 0.0996 - val_accuracy: 0.9560 - val_recall: 0.8953
Epoch 16/30: 14398/14398 [=====] - 8s 528us/step - loss: 0.0729 - accuracy: 0.9694 - recall: 0.8965
- val_loss: 0.1276 - val_accuracy: 0.9438 - val_recall: 0.8974
Epoch 17/30: 14398/14398 [=====] - 7s 513us/step - loss: 0.0707 - accuracy: 0.9706 - recall: 0.8983
- val_loss: 0.0959 - val_accuracy: 0.9575 - val_recall: 0.8993
Epoch 18/30: 14398/14398 [=====] - 7s 504us/step - loss: 0.0692 - accuracy: 0.9716 - recall: 0.9004
- val_loss: 0.0893 - val_accuracy: 0.9615 - val_recall: 0.9014
Epoch 19/30: 14398/14398 [=====] - 7s 515us/step - loss: 0.0672 - accuracy: 0.9727 - recall: 0.9024
- val_loss: 0.0967 - val_accuracy: 0.9583 - val_recall: 0.9033
Epoch 20/30: 14398/14398 [=====] - 7s 511us/step - loss: 0.0661 - accuracy: 0.9725 - recall: 0.9042
- val_loss: 0.0962 - val_accuracy: 0.9580 - val_recall: 0.9050
Epoch 21/30: 14398/14398 [=====] - 8s 531us/step - loss: 0.0643 - accuracy: 0.9732 - recall: 0.9058
- val_loss: 0.0912 - val_accuracy: 0.9606 - val_recall: 0.9066
Epoch 22/30: 14398/14398 [=====] - 8s 530us/step - loss: 0.0621 - accuracy: 0.9743 - recall: 0.9075
- val_loss: 0.0879 - val_accuracy: 0.9635 - val_recall: 0.9082
Epoch 23/30: 14398/14398 [=====] - 8s 535us/step - loss: 0.0603 - accuracy: 0.9749 - recall: 0.9090
- val_loss: 0.0881 - val_accuracy: 0.9635 - val_recall: 0.9098
Epoch 24/30: 14398/14398 [=====] - 8s 565us/step - loss: 0.0592 - accuracy: 0.9757 - recall: 0.9105
- val_loss: 0.1386 - val_accuracy: 0.9426 - val_recall: 0.9111
Epoch 25/30: 14398/14398 [=====] - 8s 541us/step - loss: 0.0580 - accuracy: 0.9766 - recall: 0.9116
- val_loss: 0.1162 - val_accuracy: 0.9532 - val_recall: 0.9122
Epoch 26/30: 14398/14398 [=====] - 8s 547us/step - loss: 0.0571 - accuracy: 0.9764 - recall: 0.9128
- val_loss: 0.1230 - val_accuracy: 0.9489 - val_recall: 0.9133
Epoch 27/30: 14398/14398 [=====] - 8s 527us/step - loss: 0.0559 - accuracy: 0.9773 - recall: 0.9138
- val_loss: 0.0893 - val_accuracy: 0.9625 - val_recall: 0.9145
Epoch 28/30: 14398/14398 [=====] - 7s 512us/step - loss: 0.0528 - accuracy: 0.9793 - recall: 0.9151
- val_loss: 0.0934 - val_accuracy: 0.9612 - val_recall: 0.9157
Epoch 29/30: 14398/14398 [=====] - 9s 593us/step - loss: 0.0517 - accuracy: 0.9798 - recall: 0.9163
- val_loss: 0.0934 - val_accuracy: 0.9626 - val_recall: 0.9170
Model: "sequential_1"
```

Layer (type)	Output Shape	Param #
dense_1 (Dense)	(None, 903)	816312
dense_2 (Dense)	(None, 600)	542400
dense_3 (Dense)	(None, 100)	60100
dense_4 (Dense)	(None, 600)	60600
dense_5 (Dense)	(None, 4)	2404
Total params: 1,481,816		
Trainable params: 1,481,816		
Non-trainable params: 0		

In [54]:

Layer (type)	Output Shape	Param #
dense_1 (Dense)	(None, 903)	816312
dense_2 (Dense)	(None, 600)	542400
dense_3 (Dense)	(None, 100)	60100
dense_4 (Dense)	(None, 600)	60600
dense_5 (Dense)	(None, 4)	2404
Total params: 1,481,816		
Trainable params: 1,481,816		
Non-trainable params: 0		

In [55]:



In [56]:

```
scores = model.evaluate(X_test_n, y_test_n, batch_size=32)
print("\n%s: %.2f%%" % (model.metrics_names[1], scores[1]*100))

check = model.evaluate(X_test_n, y_test_n, batch_size=20)
print(model.metrics_names)
print(check)
```

```
1440/1440 [=====] - 0s 121us/step
accuracy: 96.20%
1440/1440 [=====] - 0s 165us/step
('loss', accuracy, 'recall')
(0.09680599115339811, 0.9619791507720947, 0.9170161485671997)

Random Forest classification
```

In [57]:

```
from sklearn.ensemble import RandomForestClassifier
#20 estimators
r_forest = RandomForestClassifier(n_estimators=20)
r_forest.fit(X_train_n, y_train_n)
r_fit = r_forest.fit(X_train_n, y_train_n)
y_pred_rf = r_fit.predict(X_test_n)
```

In [58]:

```
print("Accuracy:", metrics.accuracy_score(y_test_n, y_pred_rf))

print("Precision:", metrics.precision_score(y_test_n, y_pred_rf, average='weighted'))

print("Recall:", metrics.recall_score(y_test_n, y_pred_rf, average='weighted'))
```

Accuracy: 0.8909722222222223
Precision: 0.9271133460397177
Recall: 0.8909722222222223

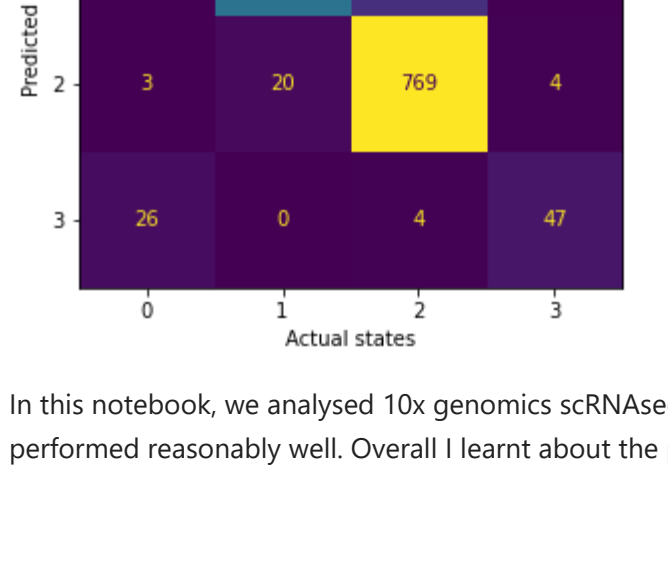
In [59]:

```
from sklearn.metrics import ConfusionMatrixDisplay

fig, ax = plt.subplots(1,1,figsize=(9,5))

ConfusionMatrixDisplay(confusion_matrix(y_test_n.argmax(axis=1), y_pred_rf.argmax(axis=1), labels=[0,1,2,3]),
                        display_labels=[0,1,2,3]).plot(values_format='.0f', ax=ax)

ax.set_title('Confusion Matrix of Random Forest Classification')
plt.ylabel('Predicted states')
plt.xlabel('Actual states')
plt.savefig("confusionmatrixRF.png")
plt.show()
```



In [60]:



Hyperparameter tuning and Cross validation

In [78]:

```
#lets also check with some hyperparameter tuning methods to further decide if we can improve our models
from sklearn.model_selection import GridSearchCV

## lets check RF first

# Hyperparameter Tuning
forest_params = [{"max_depth": list(range(1, 10)), "n_estimators": list(range(1, 10))}]

clf = GridSearchCV(r_forest, forest_params, cv = 10, scoring='accuracy')
clf.fit(X_train_n, y_train_n)
best_params = clf.best_params_
print('Best parameters:', best_params)

Best parameters: {'max_depth': 9, 'n_estimators': 8}
```

In [79]:

```
from sklearn.metrics import accuracy_score
# Fit and Evaluate on Testing Set using best params
r_forest.set_params(**best_params)
r_forest.fit(X_train_n, y_train_n)
y_pred = r_forest.predict(X_test_n)
acc = accuracy_score(y_test_n, y_pred)
print("Accuracy on test set:", acc)
```

Accuracy on test set: 0.8881944444444444

So with CV and hyperparameter tuning the accuracy is about the same

k-NN

In [61]:

```
from sklearn.neighbors import KNeighborsClassifier
knn_clf = KNeighborsClassifier()
knn_clf.fit(X_train_n, y_train_n)
ypred=knn_clf.predict(X_test_n)
```

In [62]:

```
print("Accuracy:", metrics.accuracy_score(y_test_n, ypred))

print("Precision:", metrics.precision_score(y_test_n, ypred, average='weighted'))

print("Recall:", metrics.recall_score(y_test_n, ypred, average='weighted'))
```

Accuracy: 0.8625
Precision: 0.8694161874405049
Recall: 0.8625

In [63]:

```
fig, ax = plt.subplots(1,1,figsize=(9,5))

ConfusionMatrixDisplay(confusion_matrix(y_test_n.argmax(axis=1), ypred.argmax(axis=1), labels=[0,1,2,3]),
                        display_labels=[0,1,2,3]).plot(values_format=".0f", ax=ax)

ax.set_title('Confusion Matrix of kNN Classification')
plt.ylabel('Predicted states')
plt.xlabel('Actual states')
plt.savefig("confusionmatrixkNN.png")
plt.show()
```



In this notebook, we analysed 10x genomics scRNAseq PBMCs. Using a couple of classification machine learning techniques our models performed reasonably well. Overall I learnt about the pipeline analysis for scRNAseq.)