```
import pandas as pd
import numpy as np
import keras
from keras import layers
# May need to import more things (e.g regularizers)
```

Upload the labels.csv and processed_counts.csv files to colab or your local workspace.

This data associates a cell barcode, such as "AAAGCCTGGCTAAC-1", to a certain cell type label, such as "CD14+ Monocyte". For each cell barcode, there are also log RNA seq counts of 765 different genes, such as HES4.

label.csv stores the association between a cell barcode and a cell type label.

processed_counts.csv stores the normalized log read counts for each cell, where each row represents a single cell, and each column represents a gene.

```
labels_pd = pd.read_csv("labels.csv")
counts_pd = pd.read_csv("processed_counts.csv")
```

```
labels_pd
```

```
counts_pd.rename(columns = {'Unnamed: 0':'index'}, inplace = True)
```

```
counts_pd
```

	index	HES4	TNFRSF4	SSU72	PARK7	RBP7	SRM	MAD2L2	AGTRAP
0	AAAGCCTGGCTAAC-	-0.326	-0.191	-0.728	-0.301	3.386	-0.531	2.016	3.377
3	[illegible]	-0.326	-0.191	1.134	-0.157	-0.174	-0.531	-0.451	-0.486
4	[illegible]	-0.326	-0.191	-0.728	-0.607	-0.174	-0.531	-0.451	0.787

```
counts_pd

labels_pd

labeled_counts = pd.merge(counts_pd, labels_pd, on="index")

labeled_counts
```

Shuffle your data. Make sure your labels and the counts are shuffled together.

Split into train and test sets (80:20 split)

```
# https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.ShuffleSpl
from sklearn.model_selection import ShuffleSplit

#counts_pd
#labels_pd


X = np.array([[1, 2], [3, 4], [5, 6], [7, 8], [3, 4], [5, 6]])
y = np.array([1, 2, 1, 2, 1, 2])
rs = ShuffleSplit(n_splits=1, test_size=.20, random_state=0)

rs.get_n_splits(counts_pd)

for train_index, test_index in rs.split(counts_pd,labels_pd):
  print("TRAIN:", train_index, "TEST:", test_index)
  print(train_index.shape)
  print(test_index.shape)
  break
```

```
TRAIN: [ 45 285  62 386 668 299 140 584 490 127 665 144  21 480 511  35 412 272
  77 249 333 374 263 101 279 196 294 170 460 542 409 477 691 696 214 165
 553  34 685 200 596 155 316 215 547 451 494  12 161 690 561 443 258 159
 338  78  92 476  66 261 319 283 641 271 369 540 623  15 420 247   6 475
  71 517 408 315 303 104 632 188 395 351 404 686  90 603 245 204 350 118
 293 661 218 519 250 616 415 109 375 205 339 190 327 681 439 390 495 647
 479 194 362 452 310 422 332 132 233 173 178 678 530 206  96 601 571 385
  89 355 179   0  46 171 558 570 107 505 392 223 397 102 108 213 426 649
 246 416 367 125 608 498 478 566 224  26 403 484 497   3 134 317 186 516
 295 585 552 631 225 573 568 500 458 669 434 622 342 527 112 482  20  65
 298 126 259 521 440 586 137 436   7 643 309 474 185 402 153  54  30 625
```

```
 100 652 576 664 237  56  60 496 262 535 264 653 400 208 453 391 167  38
 609 572 300 441 432 682 163 124 154 336  59 642 304 657 549 343 526 311
 413 158  51 499 462 567 361 399 425 219 308  74 282 449   4 597 541 424
 501   5 141 513 135 447 378 628  22 636 276 284 270 670 281 220 634 320
 483 120  81 687  13 419 679 672 160 699 238 379 579 485 492 523 195 191
 116 534 602 164 106  16  63 384 105 489 660 329 345 612 405 522  93 518
 433  83 591 348 648 198 145 414 150  39 514 615 640 539 322 253 627 663
 688 357  69   2 340 221 654 146 289 241  29 577 114 487 695 564 556 176
 651 372 168 347 376 531 613 598 680 189 136 446 588 254 437 382 620 509
 662 618 593 255 232 133  33  88 418 290  44 353 341  61 671 199 429 394
 297  73 393 692 583 589 217 578 421 138 212 590 587 644 234  67  24 381
 216 129 349 111 166 207 438 274 595 698 525 287 469 326 121 507 228 445
 117 464  25 110 149 152 528 461 139 260 323 630 248 450 410  19 328 296
 269 226  94 515 280 286 655 444 184 371 614 683 275 658 182  32  80 307
  11  43  86  36  58  41 411 562 209 148 594 123 574  98 377 130  23 638
 555 370 512 383 201 368 554 610 387 292 256 606 197  95 676 169 581 305
 560 373 227 143 180 131  47 324 203  84 633 565 611 398  91  82 430 119
 291  57 321 257 666 442  42 617 388 335 273 488 550  53 673 128  28 183
 459 510 151 244 543 544 639 265 288 423 147 659 177  99 448 431 115  72
 537 677 689 174  87 551 486 314 396 600 472  70 599 277   9 359 192 629
 559 684] TEST: [306 604  40 493  14 548 267  31 252 103 675 278 536 313  85 352
 529 502 545 344 624 621 582 538 520 533 619   1 546 235 172  75 532 242
 407 468 231 229 181  27 210 401 364 465 471 569 463 592 356 646 366 637
 406 312 360 142   8  79 365 175 193 354  50 656 491 607 455 605 330  10
  68 467 694 239 473 524 230 575 650 358 318 268 635 457 122 302 251 626
 157 693 389 346  37 243  48 481 454 240 162 508  76  64  52 557 334 236
 674 563 428 187 331 156 363 325 222 435 456 470 113 697  97  18 506 301
  55  49 380 466 211  17 503 202 580 337 266 645 504 417]
(560,)
(140,)
```

Create a fully connected neural network for your autoencoder. Your latent space can be of any size less than or equal to 64. Too large may result in a poor visualization, and too small may result in high loss. 32 is a good starting point.

Consider using more than 1 hidden layer, and a sparcity constraint (l1 regularization).

Have an encoder model which is a model of only the layers for the encoding.

```
#discussion model
import keras
from keras import layers
from keras import regularizers
import tensorflow as tf


encoding_dim = 32 # 32 floats (we're going from 767 dimensions -> 32 dimensions)
input_img = keras.Input(shape=(765,))

### OG
#encoded = layers.Dense(encoding_dim, activation='relu',activity_regularizer=regulari:
#decoded = layers.Dense(765, activation='sigmoid')(encoded)
```

```
### OG



### new
encoded = layers.Dense(128, activation='relu', activity_regularizer=regularizers.l1(1(
#encoded = layers.Dense(64, activation='relu', activity_regularizer=regularizers.l1(1(
encoded = layers.Dense(32, activation='relu', activity_regularizer=regularizers.l1(10e

decoded = layers.Dense(32, activation='relu')(encoded)
#decoded = layers.Dense(128, activation='relu')(decoded)
decoded = layers.Dense(765, activation='sigmoid')(decoded)
##### new

autoencoder = keras.Model(input_img, decoded)
encoder = keras.Model(input_img, encoded)

autoencoder.compile(optimizer='adam', loss=tf.keras.losses.MeanSquaredError())


train_index = train_index.tolist()
test_index = test_index.tolist()


counts_pd_train = counts_pd.loc[train_index]
counts_pd_test = counts_pd.loc[test_index]

labels_pd_train = labels_pd.loc[train_index]
labels_pd_test = labels_pd.loc[test_index]


counts_pd_train_np = counts_pd_train.copy().to_numpy()
counts_pd_test_np = counts_pd_test.copy().to_numpy()


counts_pd_train_np.shape
```

```
(560, 765)
```

```
#remove the first column
#counts_pd_train_nl = np.delete(counts_pd_train_np,[0],1)
#counts_pd_test_nl = np.delete(counts_pd_test_np,[0],1)


#cleaned up data ready to be put through the autoencoder
counts_pd_train_nl = counts_pd_train_np.astype('float32')#/255
counts_pd_test_nl = counts_pd_test_np.astype('float32')#/255

counts_pd_test_nl.shape
```

```
(140, 765)
```

```
autoencoder.fit(counts_pd_train_nl, counts_pd_train_nl,
                epochs=5, #if redone train for just 20-30
                batch_size=256,
                shuffle=True,
                validation_data=(counts_pd_test_nl, counts_pd_test_nl))
```

```
Epoch 1/5
3/3 [==============================] - 0s 134ms/step - loss: 0.9954 - val_loss: 0
Epoch 2/5
3/3 [==============================] - 0s 55ms/step - loss: 0.9923 - val_loss: 0
Epoch 3/5
3/3 [==============================] - 0s 33ms/step - loss: 0.9890 - val_loss: 0
Epoch 4/5
3/3 [==============================] - 0s 29ms/step - loss: 0.9856 - val_loss: 0
Epoch 5/5
3/3 [==============================] - 0s 56ms/step - loss: 0.9823 - val_loss: 0
<keras.callbacks.History at 0x7ff7f08dd310>
```

```
#labels_pd_test['bulk_labels']
counts_pd_test_nl.shape
```

```
(140, 765)
```

```
#run on the test data
test_loss = autoencoder.evaluate(counts_pd_test_nl, counts_pd_test_nl, verbose=2)


print('\nTest loss:', test_loss)
```

```
5/5 - 0s - loss: 0.9713 - 34ms/epoch - 7ms/step

Test loss: 0.9713215231895447
```

Train your autoencoding using MSE loss.

Finally, identify the parameters which don't overfit, and use the same model architecture and train on all of the data together.

With a latent space size of 32, aim for 0.9 MSE loss on your test set, 0.95 with regularization. You will not be graded strictly on a loss cutoff.

```
#once didnt overfit, combine training and test, same autoenchoder just all the data
```

Use PCA and t-SNE on the dataset.

Then use PCA on the latent space representation of the dataset.

Plot all of these.

```
from sklearn.decomposition import PCA
from sklearn.manifold import TSNE

import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns


#counts_pd = counts_pd.drop(columns=['index'])
numpy_counts = counts_pd.to_numpy()



# See HW7 (part 1)
#PCA on the dataset

pca = PCA(n_components=2)
counts_pca = pca.fit_transform(numpy_counts)


plt.figure(figsize=(10,10))

sns.scatterplot(
    x=counts_pca[:,0], y=counts_pca[:,1],
    hue=labels_pd['bulk_labels'],
    alpha=0.75)

plt.show()
```

```
# Carry out t-SNE on X
tsne = TSNE(n_components=2)
counts_tsne = tsne.fit_transform(numpy_counts)
```

```
/usr/local/lib/python3.7/dist-packages/sklearn/manifold/_t_sne.py:783: FutureWar
  FutureWarning,
/usr/local/lib/python3.7/dist-packages/sklearn/manifold/_t_sne.py:793: FutureWar
  FutureWarning,
```

```
plt.figure(figsize=(10,10))

sns.scatterplot(
    x=counts_tsne[:,0], y=counts_tsne[:,1],
    hue=labels_pd['bulk_labels'],
    alpha=0.75)

plt.show()
```

Compare the results of PCA, t-SNE, and your autoencoder as ways to visualize the data.

```
# Use Model.predict() to retrieve the embedding
# Use PCA on embedding
# Use tsne on embedding
```

```
### Get from encoder --> goes down to 32 dimensions
encoded_counts = encoder.predict(numpy_counts)

#PCA on the encoded output
pca = PCA(n_components=2)
en_counts_pca = pca.fit_transform(encoded_counts)

#tsne on the encoded output
tsne = TSNE(n_components=2)
en_counts_tsne = tsne.fit_transform(encoded_counts)
```

```
/usr/local/lib/python3.7/dist-packages/sklearn/manifold/_t_sne.py:783: FutureWar
  FutureWarning,
/usr/local/lib/python3.7/dist-packages/sklearn/manifold/_t_sne.py:793: FutureWar
  FutureWarning,
```

```
#encoded_counts.shape
en_counts_pca.shape
```

```
(700, 2)
```

```
plt.figure(figsize=(10,10))

sns.scatterplot(
    x=en_counts_pca[:,0], y=en_counts_pca[:,1],
    hue=labels_pd['bulk_labels'],
    alpha=0.75)

plt.show()
```

```
en_counts_tsne.shape
```

```
(700, 2)
```

```
plt.figure(figsize=(10,10))

sns.scatterplot(
    x=en_counts_tsne[:,0], y=en_counts_tsne[:,1],
    hue=labels_pd['bulk_labels'],
    alpha=0.75)

plt.show()
```

```
#prep all the data

counts_np = counts_pd.copy().to_numpy()

counts_pd_t = counts_np.astype('float32')


#train on entire dataset
autoencoder.fit(counts_pd_t, counts_pd_t,
                epochs=25, #if redone train for just 20-30
                batch_size=256,
                shuffle=True)
```

```
Epoch 1/25
3/3 [==============================] - 1s 9ms/step - loss: 1.2615
Epoch 2/25
3/3 [==============================] - 0s 8ms/step - loss: 1.2537
Epoch 3/25
3/3 [==============================] - 0s 8ms/step - loss: 1.2447
Epoch 4/25
3/3 [==============================] - 0s 8ms/step - loss: 1.2304
Epoch 5/25
3/3 [==============================] - 0s 9ms/step - loss: 1.2084
Epoch 6/25
3/3 [==============================] - 0s 8ms/step - loss: 1.1775
Epoch 7/25
3/3 [==============================] - 0s 8ms/step - loss: 1.1397
```

```
Epoch 8/25
3/3 [==============================] - 0s 8ms/step - loss: 1.0991
Epoch 9/25
3/3 [==============================] - 0s 7ms/step - loss: 1.0633
Epoch 10/25
3/3 [==============================] - 0s 7ms/step - loss: 1.0382
Epoch 11/25
3/3 [==============================] - 0s 8ms/step - loss: 1.0242
Epoch 12/25
3/3 [==============================] - 0s 10ms/step - loss: 1.0176
Epoch 13/25
3/3 [==============================] - 0s 8ms/step - loss: 1.0138
Epoch 14/25
3/3 [==============================] - 0s 7ms/step - loss: 1.0106
Epoch 15/25
3/3 [==============================] - 0s 8ms/step - loss: 1.0070
Epoch 16/25
3/3 [==============================] - 0s 8ms/step - loss: 1.0030
Epoch 17/25
3/3 [==============================] - 0s 8ms/step - loss: 0.9992
Epoch 18/25
3/3 [==============================] - 0s 8ms/step - loss: 0.9953
Epoch 19/25
3/3 [==============================] - 0s 8ms/step - loss: 0.9913
Epoch 20/25
3/3 [==============================] - 0s 7ms/step - loss: 0.9873
Epoch 21/25
3/3 [==============================] - 0s 11ms/step - loss: 0.9835
Epoch 22/25
3/3 [==============================] - 0s 8ms/step - loss: 0.9798
Epoch 23/25
3/3 [==============================] - 0s 8ms/step - loss: 0.9762
Epoch 24/25
3/3 [==============================] - 0s 7ms/step - loss: 0.9727
Epoch 25/25
3/3 [==============================] - 0s 7ms/step - loss: 0.9692
<keras.callbacks.History at 0x7ff7f066e510>
```

```
#export the final encoded data
encoded_counts_final = encoder.predict(numpy_counts)
```

```
df = pd.DataFrame(encoded_counts_final)
```

```
df.to_csv('encoded_counts.csv')
```

```
import pandas as pd
import numpy as np
import keras
from keras import layers
```

Upload the labels.csv and processed_counts.csv files to colab or your local workspace.

Copied from Part 1: This data associates a cell barcode, such as "AAAGCCTGGCTAAC-1", to a certain cell type label, such as "CD14+ Monocyte". For each cell barcode, there are also log RNA seq counts of 765 different genes, such as HES4.

label.csv stores the association between a cell barcode and a cell type label.

processed_counts.csv stores the normalized log read counts for each cell, where each row represents a single cell, and each column represents a gene.

```
labels_pd = pd.read_csv("labels.csv")
counts_pd = pd.read_csv("processed_counts.csv")
processed_pd = pd.read_csv("encoded_counts.csv")
```

```
df1 = processed_pd.merge(labels_pd,left_index=True, right_index=True)
df1.drop("Unnamed: 0", axis=1, inplace=True)
df1.drop("index", axis=1, inplace=True)

df1
```

	0	1	2	3	4	5	6	7	8	
0	0.000000	0.000000	0.000000	8.503382	0.000000	0.0	0.000000	0.0	0.000000	12.0071

```
labels_pd.index = labels_pd['index']
labels_pd.drop("index", axis=1, inplace=True)
counts_pd.index = counts_pd['Unnamed: 0']
counts_pd.drop("Unnamed: 0", axis=1, inplace=True)

df = counts_pd.merge(labels_pd, left_index=True, right_index=True).dropna()
df
```

One-hot encode the cell-type.

Shuffle your data. Make sure your labels and the counts are shuffled together.

Split into train and test sets (80:20 split)

```
#splitting up the total bulk data
categories = df['bulk_labels'].unique()
print(categories)

#one-hot encoding
y = np.zeros((len(df), len(categories)))
for i in range(len(df)):
    cell_type = df.iloc[i]['bulk_labels']
    pos = np.where(categories == cell_type)[0]
    y[i, pos] = 1

#remove label when processing input data
X = df.drop('bulk_labels', axis=1).values

#shufle and 80:20 split
np.random.seed(100)
permutation = np.random.permutation(len(X))
X, y = X[permutation], y[permutation]

X_train, y_train = X[:int(len(X)*0.8)], y[:int(len(y)*0.8)]
X_test, y_test = X[int(len(X)*0.8):], y[int(len(y)*0.8):]
```

```
['CD14+ Monocyte' 'Dendritic' 'CD56+ NK' 'CD4+/CD25 T Reg' 'CD19+ B'
 'CD8+ Cytotoxic T' 'CD4+/CD45RO+ Memory' 'CD8+/CD45RA+ Naive Cytotoxic'
 'CD4+/CD45RA+/CD25- Naive T' 'CD34+']
```

```
#splitting up the enbedded data
categories = df1['bulk_labels'].unique()
print(categories)

#one-hot encoding
y = np.zeros((len(df1), len(categories)))
for i in range(len(df1)):
    cell_type = df1.iloc[i]['bulk_labels']
    pos = np.where(categories == cell_type)[0]
    y[i, pos] = 1

#remove label when processing input data
X = df1.drop('bulk_labels', axis=1).values

encoded_clean = X
encoded_clean_labels = y

#shufle and 80:20 split
np.random.seed(100)
permutation = np.random.permutation(len(X))
X, y = X[permutation], y[permutation]
```

```
X_train1, y_train1 = X[:int(len(X)*0.8)], y[:int(len(y)*0.8)]
X_test1, y_test1 = X[int(len(X)*0.8):], y[int(len(y)*0.8):]
```

```
['CD14+ Monocyte' 'Dendritic' 'CD56+ NK' 'CD4+/CD25 T Reg' 'CD19+ B'
 'CD8+ Cytotoxic T' 'CD4+/CD45RO+ Memory' 'CD8+/CD45RA+ Naive Cytotoxic'
```

```
#print(X_train)
#print(y_train) #one hot encoded for different cell types

print(encoded_clean.shape)
print(encoded_clean_labels.shape)
```

```
(700, 32)
(700, 10)
```

```
#Visualize the One-hot encoded Prediction Labels
import matplotlib.pyplot as plt
plt.figure(figsize=(9,3), dpi=300)
plt.imshow(y_train[:50])
```

Apply classification algorithms to the training data, tune on validation data (if present), and evaluate on test data.

You can also apply classification downstream of last week's autoencoder latent space representation.

```
# Decision trees: https://scikit-learn.org/stable/modules/tree.html
# Ensemble methods: https://scikit-learn.org/stable/modules/ensemble.html
```

```
(560, 765)
```

```
#Decision Tree####
##################

#have to be mindful of decision trees as they tend to overfit to theirparituclar data
#and if you had variance to a dataset you are likley to get an entirly different tree

#Could do PCA before on all the data or just use the embedded data

from sklearn import tree
#use embedding for the decision tree and ensemble method

from sklearn.model_selection import cross_val_score
from sklearn.tree import DecisionTreeClassifier
# 3,5 ,5

clf = DecisionTreeClassifier(random_state=0,max_depth = 20, min_samples_split = 7, mir

# max depth is how deep the tree is allowed to be (lower)
```

```
# min_samples split is how many samples at each split node,  (higher)
# min_samples_leaf is how many samples per leaf --> if low then very complex. (higher)

cross_val_score(clf, encoded_clean, encoded_clean_labels, cv=5).mean()
```

```
0.667142857142857
```

```
#random forest####
#################


#randomly pick a subset of features to work with
from sklearn.ensemble import RandomForestClassifier


#n estimators is the number of decision trees you are spitting up
#max_features = 'sqrt',5 --> lower their will be less variance
#min_samples_split


clf = RandomForestClassifier(n_estimators=15)#,min_samples_split=2)#,max_features = 5)
clf = clf.fit(X_train1,y_train1)

#take average of predctions

clf.score(X_test1, y_test1)
```

```
0.7071428571428572
```

```
#Ensemble bagging and random forest#
####################################

#average the predictions for multiple models
#each working with a subset of the data

from sklearn.ensemble import BaggingClassifier

#max samples is what proportion of the data is it going to train on
#max_features is the max num of features each will train on

bagging = BaggingClassifier(clf, max_samples=0.5, max_features=0.5)


clf = clf.fit(X_train1,y_train1)
clf.score(X_test1, y_test1)
```

```
0.7357142857142858
```

```
# FFNN hints:

# Use softmax at the end, reLU for the rest
# Add layers until desired loss

# Categorical cross-entropy for loss func

# Add dropout layers to avoid overfitting

# Can also do bagging with FFNNs (but probably not necessary): https://machinelearning

#use all the data here
import keras
from keras import layers
from keras import regularizers
import tensorflow as tf

model = keras.Sequential()

model.add(tf.keras.Input(shape=(765)))


model.add(tf.keras.layers.Dense(100, input_dim=32, activation='relu'))
model.add(tf.keras.layers.Dropout(0.3))


model.add(tf.keras.layers.Dense(100, input_dim=32, activation='relu'))
model.add(tf.keras.layers.Dropout(0.3))

#model.add(tf.keras.layers.Dense(200, input_dim=32, activation='relu'))
#model.add(tf.keras.layers.Dropout(0.2))

model.add(tf.keras.layers.Dense(100, input_dim=32, activation='relu'))
model.add(tf.keras.layers.Dropout(0.3))


model.add(tf.keras.layers.Dense(10, activation='softmax'))
model.compile(loss="categorical_crossentropy", optimizer='adam', metrics=['accuracy'])



model.fit(X_train, y_train, epochs=15)
```

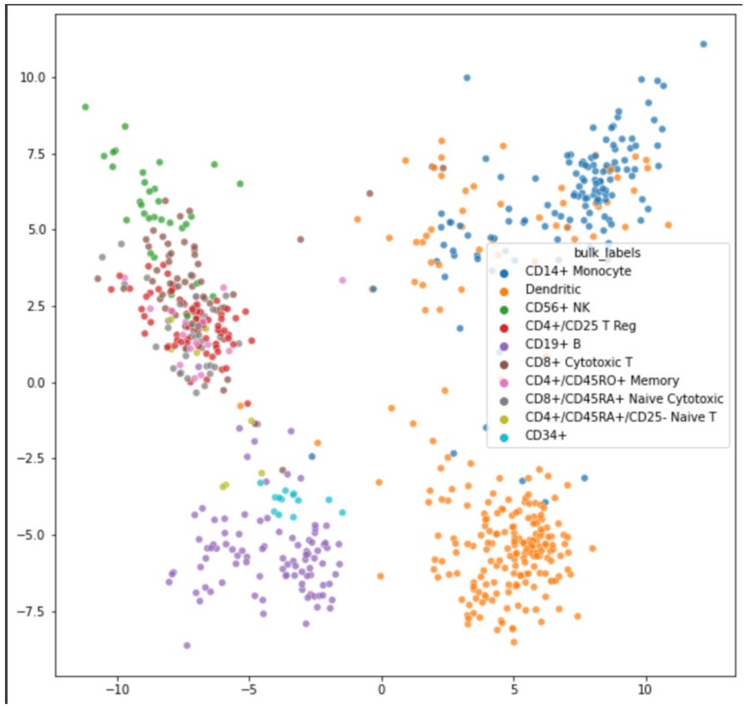
```
Epoch 1/15
18/18 [==============================] - 1s 4ms/step - loss: 2.1427 - accuracy: 
Epoch 2/15
18/18 [==============================] - 0s 4ms/step - loss: 1.3655 - accuracy: 
Epoch 3/15
18/18 [==============================] - 0s 4ms/step - loss: 1.0768 - accuracy: 
Epoch 4/15
```

```
18/18 [==============================] - 0s 4ms/step - loss: 0.8016 - accuracy: 
Epoch 5/15
18/18 [==============================] - 0s 4ms/step - loss: 0.6433 - accuracy: 
Epoch 6/15
18/18 [==============================] - 0s 4ms/step - loss: 0.5296 - accuracy: 
Epoch 7/15
18/18 [==============================] - 0s 4ms/step - loss: 0.4043 - accuracy: 
Epoch 8/15
18/18 [==============================] - 0s 4ms/step - loss: 0.3434 - accuracy: 
Epoch 9/15
18/18 [==============================] - 0s 4ms/step - loss: 0.2942 - accuracy: 
Epoch 10/15
18/18 [==============================] - 0s 5ms/step - loss: 0.2411 - accuracy: 
Epoch 11/15
18/18 [==============================] - 0s 4ms/step - loss: 0.1919 - accuracy: 
Epoch 12/15
18/18 [==============================] - 0s 4ms/step - loss: 0.1982 - accuracy: 
Epoch 13/15
18/18 [==============================] - 0s 4ms/step - loss: 0.1431 - accuracy: 
Epoch 14/15
18/18 [==============================] - 0s 4ms/step - loss: 0.1366 - accuracy: 
Epoch 15/15
18/18 [==============================] - 0s 3ms/step - loss: 0.1050 - accuracy: 
<keras.callbacks.History at 0x7fb983f67d50>
```

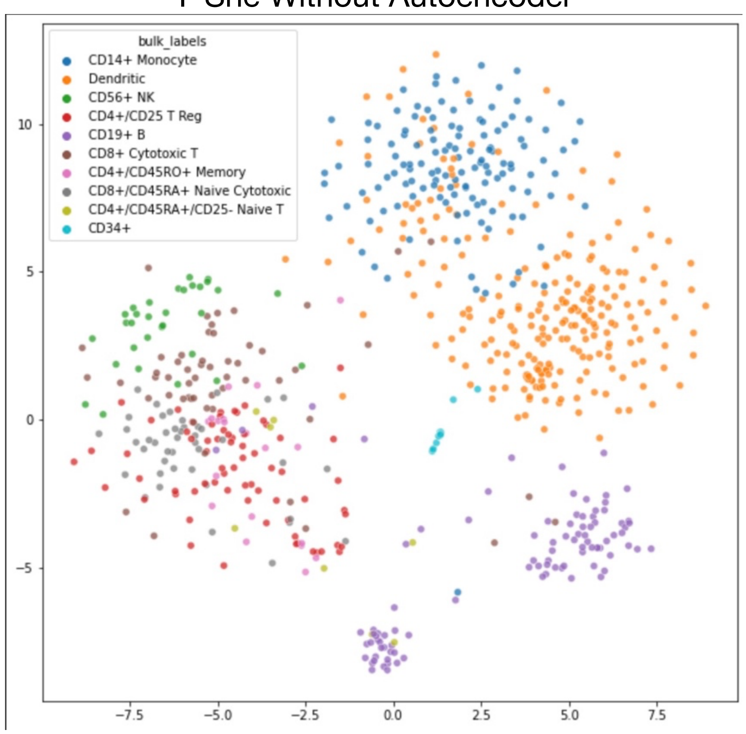
```
# evaluate the model

#X_train, y_train
#X_test, y_test


#0.9 achevable

_, train_acc = model.evaluate(X_train, y_train, verbose=0)
_, test_acc = model.evaluate(X_test, y_test, verbose=0)
print('Train: %.3f, Test: %.3f' % (train_acc, test_acc))
```

```
Train: 1.000, Test: 0.843
```

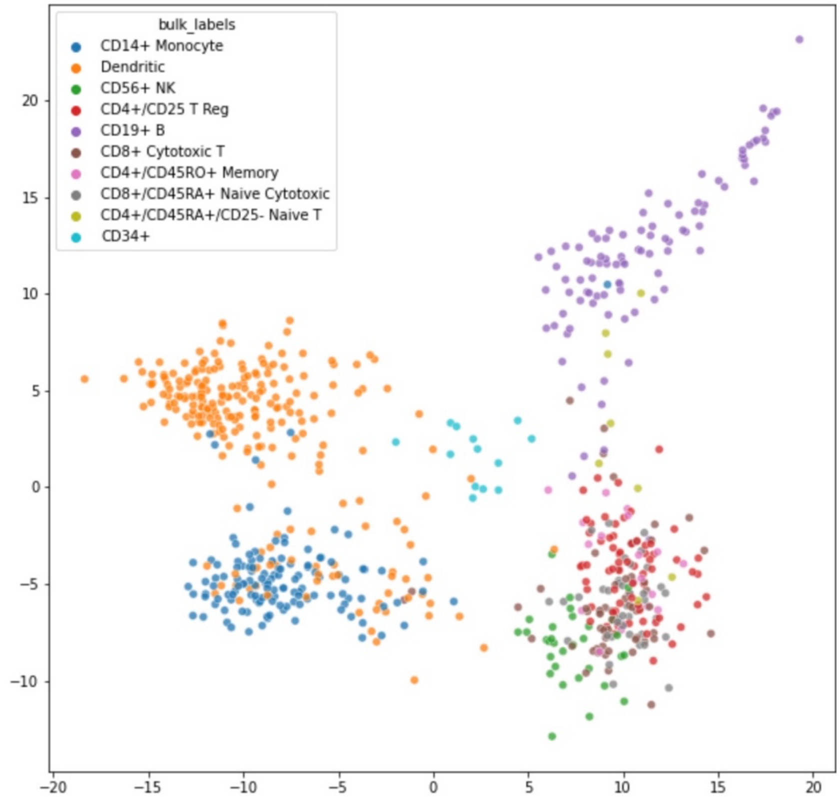
PCA – Before Autoencoder



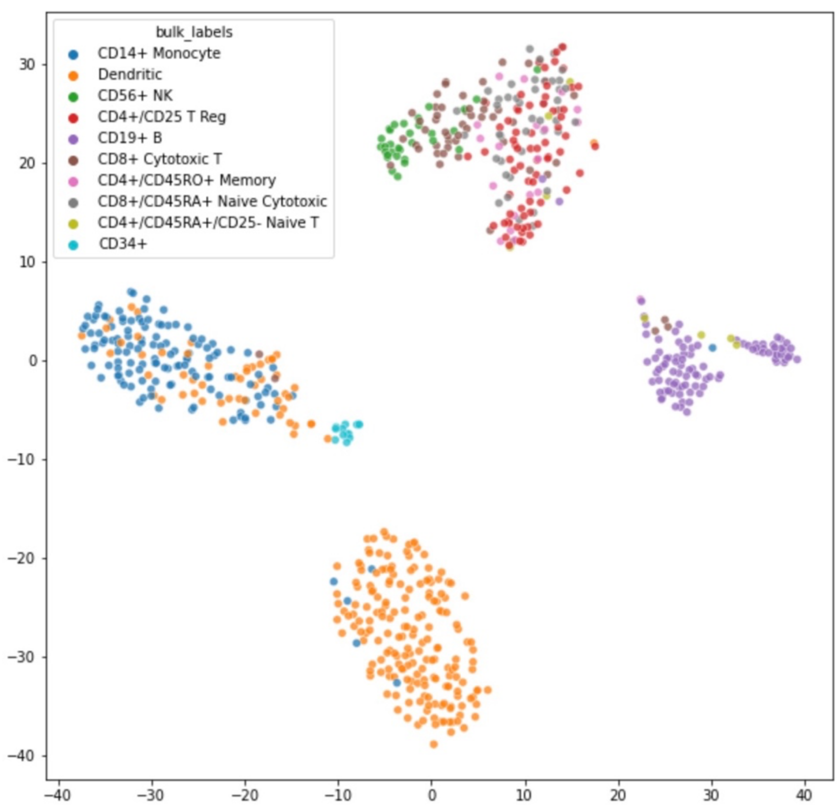
T-Sne Without Autoencoder



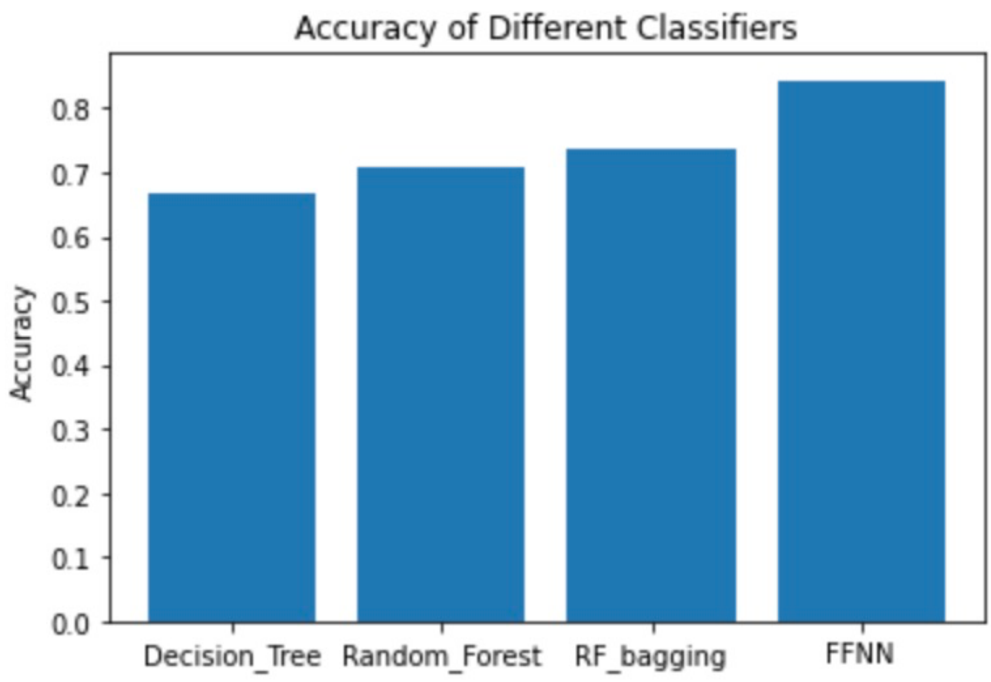
PCA – With Autoencoder



T-Sne with autoencoder



PART 1: Write up
The approaches I took for dimensionality reduction were PCA, T-Sne and an autoencoder. In terms of the approaches ranked from most effective to least effective I believe the autoencoder was the most effective, then the PCA then T-Sne. Looking at the autoencoder graphs in both PCA and t-Sne representations there were clearer clustering of the CD19+ B, dendritic cells and CD14+ Monocytes. Compared to the PCA and T-sne representations done before autoencoding. Looking even closer that the autoencoded data in t-sne, this graph has the clearer clustering out of all of the dimensionality reduced data. Then between the PCA and T-sne figures without autoencoding, PCA has tighter more defined clusters.



PART 2: Write up
The four classification methods I used to try and classify the data back to cell types were a decision tree with cross-validation, random forest, random forest with bagging and finally a feed-forward neural network. The accuracy is in order from left to right is 66.7%, 70.7%, 73.6% and 84.3%. I first started with a decision tree. I found that after trial and error that the optimal parameters were a max depth of 20, with min_samples_split and min_samples_leaf at 7. Min samples split represents the number of samples at each split and min samples leaf represented the number of samples in each leaf of the tree. I kept these parameters low as I did not want the classifier to be too specific and overtrain. I also did cross validation, where I trained multiple models, 5 in my case, and take the mean of the accuracies as the final accuracy score for the decision tree. Moving onto random forest which I found to be more effective then the decisions tree, I ended up using 15 estimators, which represented 15 different models. Next I implemented a random forest with bagging, which created additional data for training through using combinations with repetitions of data. This method improved random forest by 3%. But by far

the most effective classifier was the feedforward network with an overall accuracy of 84%. The model I ended up choosing was primarily using layers of dense and dropout layers. Dropout layers helping to prevent overfitting, and this was key to improving accuracy. For the last step I used a softmax activation function to instead of a relu because this model was not doing a binary classification but a classification of many cell types. Overall I found that the FFNN was the most effective model overall and is therefore the classifier I would use.