AML Homework 3

April 1, 2022

1 Homework 3

1.1 Part 1: Imbalanced Dataset

This part of homework helps you practice to classify a highly imbalanced dataset in which the number of examples in one class greatly outnumbers the examples in another. You will work with the Credit Card Fraud Detection dataset hosted on Kaggle. The aim is to detect a mere 492 fraudulent transactions from 284,807 transactions in total.

1.1.1 Instructions

Please push the .ipynb, .py, and .pdf to Github Classroom prior to the deadline. Please include your UNI as well.

Due Date : TBD

1.1.2 Davit Barblishvili

1.1.3 DB3230

1.2 0 Setup

[1]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns

import sklearn
from sklearn.metrics import confusion_matrix
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from imblearn.pipeline import make_pipeline as imb_make_pipeline
from imblearn.under_sampling import RandomUnderSampler
from imblearn.over_sampling import RandomOverSampler
from imblearn.over_sampling import SMOTE
```

1.3 1 Data processing and exploration

Download the Kaggle Credit Card Fraud data set. Features V1, V2, … V28 are the principal components obtained with PCA, the only features which have not been transformed with PCA are

'Time' and 'Amount'. Feature 'Time' contains the seconds elapsed between each transaction and the first transaction in the dataset. The feature 'Amount' is the transaction Amount, this feature can be used for example-dependant cost-sensitive learning. Feature 'Class' is the response variable and it takes value 1 in case of fraud and 0 otherwise.

[2]:
```
raw_df = pd.read_csv('https://storage.googleapis.com/download.tensorflow.org/
 ↪data/creditcard.csv')
raw_df.head()
```

[2]:
```
   Time        V1        V2        V3        V4        V5        V6        V7  \
0   0.0 -1.359807 -0.072781  2.536347  1.378155 -0.338321  0.462388  0.239599
1   0.0  1.191857  0.266151  0.166480  0.448154  0.060018 -0.082361 -0.078803
2   1.0 -1.358354 -1.340163  1.773209  0.379780 -0.503198  1.800499  0.791461
3   1.0 -0.966272 -0.185226  1.792993 -0.863291 -0.010309  1.247203  0.237609
4   2.0 -1.158233  0.877737  1.548718  0.403034 -0.407193  0.095921  0.592941

         V8        V9  …       V21       V22       V23       V24       V25  \
0  0.098698  0.363787  … -0.018307  0.277838 -0.110474  0.066928  0.128539
1  0.085102 -0.255425  … -0.225775 -0.638672  0.101288 -0.339846  0.167170
2  0.247676 -1.514654  …  0.247998  0.771679  0.909412 -0.689281 -0.327642
3  0.377436 -1.387024  … -0.108300  0.005274 -0.190321 -1.175575  0.647376
4 -0.270533  0.817739  … -0.009431  0.798278 -0.137458  0.141267 -0.206010

        V26       V27       V28  Amount  Class
0 -0.189115  0.133558 -0.021053  149.62      0
1  0.125895 -0.008983  0.014724    2.69      0
2 -0.139097 -0.055353 -0.059752  378.66      0
3 -0.221929  0.062723  0.061458  123.50      0
4  0.502292  0.219422  0.215153   69.99      0

[5 rows x 31 columns]
```

1.3.1 1.1 Examine the class label imbalance

Let's look at the dataset imbalance:

Q1. How many observations are there in this dataset? How many of them have positive label (labeled as 1)?

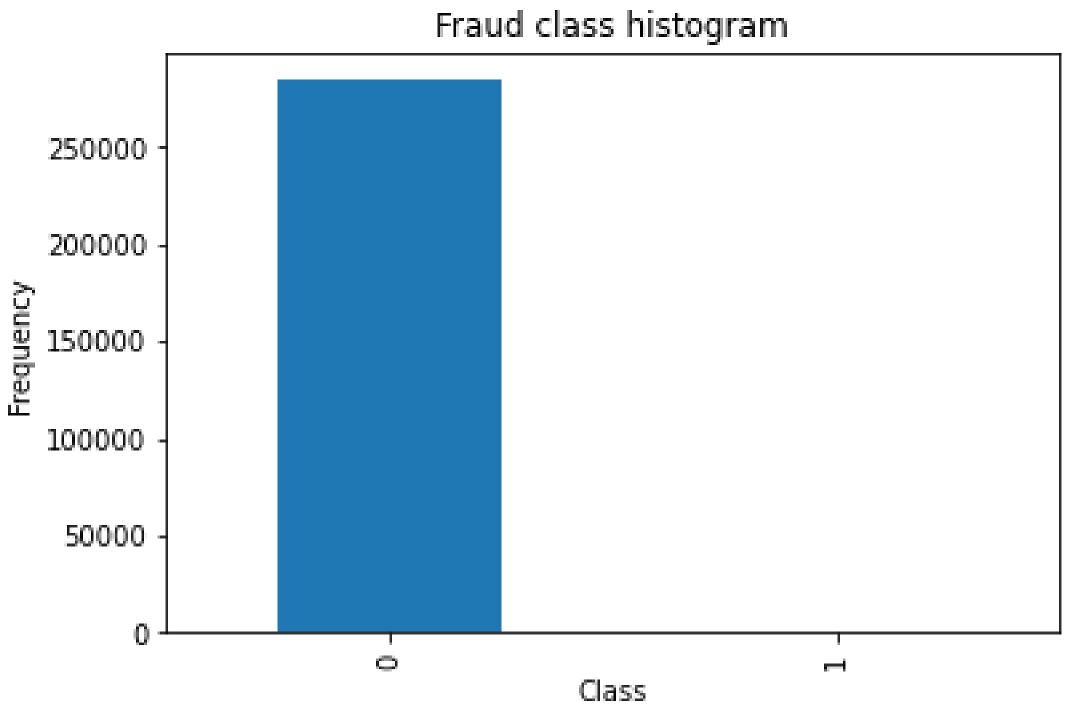
[3]:
```
# Your Code Here
import collections
target = raw_df.values[:,-1]
counter = collections.Counter(target)
for k,v in counter.items():
    per = v / len(target) * 100
    print('Class=%d, Count=%d, Percentage=%.3f%%' % (k, v, per))
```

```
Class=0, Count=284315, Percentage=99.827%
Class=1, Count=492, Percentage=0.173%
```

```
[4]: print(f'Classes Count in Credit card Fraud Dataset \n', pd.
      ↪value_counts(raw_df['Class'], sort = True).sort_index())
     credit_classes = pd.value_counts(raw_df['Class'], sort = True).sort_index()
     credit_classes.plot(kind = 'bar')
     plt.title("Fraud class histogram")
     plt.xlabel("Class")
     plt.ylabel("Frequency")
```

```
Classes Count in Credit card Fraud Dataset
 0    284315
1       492
Name: Class, dtype: int64
```

```
[4]: Text(0, 0.5, 'Frequency')
```



2 Answer

From the previous result, we can see that there are two observations, or in other words two different classes. First class is '0' which is a majority class and second one is class '1' which is minority class. We have very skewed dataset. Only 0.173% of total data belongs to class '1' which is not a data distribution we should have before doing any modeling.

2.0.1 1.2 Clean, split and normalize the data

The raw data has a few issues. First the `Time` and `Amount` columns are too variable to use directly. Drop the `Time` column (since it's not clear what it means) and take the log of the `Amount` column to reduce its range.

[5]:
```
cleaned_df = raw_df.copy()

# You don't want the `Time` column.
cleaned_df.pop('Time')

# The `Amount` column covers a huge range. Convert to log-space.
eps = 0.001 # 0 => 0.1¢
cleaned_df['Log Ammount'] = np.log(cleaned_df.pop('Amount')+eps)
```

[6]:
```
cleaned_df
```

[6]:
```
               V1         V2        V3        V4        V5        V6  \
0       -1.359807  -0.072781  2.536347  1.378155 -0.338321  0.462388
1        1.191857   0.266151  0.166480  0.448154  0.060018 -0.082361
2       -1.358354  -1.340163  1.773209  0.379780 -0.503198  1.800499
3       -0.966272  -0.185226  1.792993 -0.863291 -0.010309  1.247203
4       -1.158233   0.877737  1.548718  0.403034 -0.407193  0.095921
…             …          …         …         …         …         …
284802 -11.881118  10.071785 -9.834783 -2.066656 -5.364473 -2.606837
284803  -0.732789  -0.055080  2.035030 -0.738589  0.868229  1.058415
284804   1.919565  -0.301254 -3.249640 -0.557828  2.630515  3.031260
284805  -0.240440   0.530483  0.702510  0.689799 -0.377961  0.623708
284806  -0.533413  -0.189733  0.703337 -0.506271 -0.012546 -0.649617

              V7        V8        V9       V10  …       V21       V22  \
0       0.239599  0.098698  0.363787  0.090794  … -0.018307  0.277838
1      -0.078803  0.085102 -0.255425 -0.166974  … -0.225775 -0.638672
2       0.791461  0.247676 -1.514654  0.207643  …  0.247998  0.771679
3       0.237609  0.377436 -1.387024 -0.054952  … -0.108300  0.005274
4       0.592941 -0.270533  0.817739  0.753074  … -0.009431  0.798278
…            …         …         …         …  …         …         …
284802 -4.918215  7.305334  1.914428  4.356170  …  0.213454  0.111864
284803  0.024330  0.294869  0.584800 -0.975926  …  0.214205  0.924384
284804 -0.296827  0.708417  0.432454 -0.484782  …  0.232045  0.578229
284805 -0.686180  0.679145  0.392087 -0.399126  …  0.265245  0.800049
284806  1.577006 -0.414650  0.486180 -0.915427  …  0.261057  0.643078

             V23       V24       V25       V26       V27       V28  Class  \
0      -0.110474  0.066928  0.128539 -0.189115  0.133558 -0.021053      0
1       0.101288 -0.339846  0.167170  0.125895 -0.008983  0.014724      0
2       0.909412 -0.689281 -0.327642 -0.139097 -0.055353 -0.059752      0
3      -0.190321 -1.175575  0.647376 -0.221929  0.062723  0.061458      0
```

```
4      -0.137458  0.141267 -0.206010  0.502292  0.219422  0.215153      0
…          …        …        …        …        …        …     …
284802  1.014480 -0.509348  1.436807  0.250034  0.943651  0.823731      0
284803  0.012463 -1.016226 -0.606624 -0.395255  0.068472 -0.053527      0
284804 -0.037501  0.640134  0.265745 -0.087371  0.004455 -0.026561      0
284805 -0.163298  0.123205 -0.569159  0.546668  0.108821  0.104533      0
284806  0.376777  0.008797 -0.473649 -0.818267 -0.002415  0.013649      0

        Log Ammount
0          5.008105
1          0.989913
2          5.936641
3          4.816249
4          4.248367
…             …
284802    -0.260067
284803     3.210481
284804     4.217756
284805     2.302685
284806     5.379902

[284807 rows x 30 columns]
```

Q2. Split the dataset into development and test sets. Please set test size as 0.2 and random state as 42.

[7]:
```python
# Your Code Here
X=cleaned_df.drop(['Class'],axis=1)
y=cleaned_df['Class']

X_dev,X_test,y_dev,y_test=train_test_split(X,y,test_size=0.20,random_state=42)
```

Q3. Normalize the input features using the sklearn StandardScaler. Print the shape of your development features and test features.

[8]:
```python
# Your Code Here
scaler = StandardScaler()
X_dev = scaler.fit_transform(X_dev)
X_test = scaler.transform(X_test)
```

[9]:
```python
print(X_dev.shape)
print(X_test.shape)
```

```
(227845, 29)
(56962, 29)
```

2.0.2 1.3 Define the model and metrics

Q4. First, fit a default logistic regression model. Print the AUC and average precision of 5-fold cross validation.

```
[10]: # Your Code Here
      from sklearn.linear_model import LogisticRegression

      logreg=LogisticRegression()
      logreg.fit(X_dev,y_dev)

      y_pred_default=logreg.predict_proba(X_test)[::,1]
      y_pred_default_cfmatrix=logreg.predict(X_test)
      y_pred_default
```

```
[10]: array([9.99999919e-01, 7.06929407e-05, 4.73665958e-05, …,
             4.44154105e-04, 6.59102152e-05, 8.47817442e-04])
```

```
[11]: from sklearn import metrics


      #y_pred_default = logreg.predict_proba(X_test)[::,1]
      auc_default_legreg = metrics.roc_auc_score(y_test, y_pred_default)
      fpr_default, tpr_default, _ = metrics.roc_curve(y_test,  y_pred_default)
      print(auc_default_legreg)
```

```
0.9764750195238477
```

```
[12]: import numpy as np
      import pandas as pd
      from sklearn.model_selection import KFold
      from sklearn.model_selection import cross_val_score
      from sklearn.linear_model import LogisticRegression
      from sklearn.metrics import confusion_matrix
      from sklearn.metrics import classification_report
```

```
[13]: kfold = KFold(n_splits=5, random_state=42, shuffle=True)
      model = LogisticRegression(solver='liblinear')
      results = cross_val_score(model, X, y, cv=kfold, scoring="average_precision")
      cv_leg_reg_default_mean = results.mean()*100.0
      cv_leg_reg_default_std = results.std()*100.0

      # Output the accuracy. Calculate the mean and std across all folds.
      print("Accuracy: %.3f%% (%.3f%%)" % (cv_leg_reg_default_mean,␣
       ↪cv_leg_reg_default_std))
```

```
Accuracy: 76.134% (4.263%)
```

Q5.1. Perform random under sampling on the development set. What is the shape of your development features? How many positive and negative labels are there in

your development set? (Please set random state as 42 when performing random under sampling)

[14]:
```
# Your Code Here

rus = RandomUnderSampler(random_state=42)
# resampling X, y
X_rus, y_rus = rus.fit_resample(X_dev, y_dev)
# new class distribution
print(collections.Counter(y_rus))
```

```
Counter({0: 394, 1: 394})
```

[15]:
```
print(X_rus.shape)
print(y_rus.shape)
```

```
(788, 29)
(788,)
```

3 Answer

From the above operation, we can see that we have 50/50 distribution of positive (1) and negative (0) classes which I think is much better than what we originally had.

Q5.2. Fit a default logistic regression model using under sampling. Print the AUC and average precision of 5-fold cross validation. (Please set random state as 42 when performing random under sampling)

[16]:
```
logreg=LogisticRegression()
logreg.fit(X_rus,y_rus)
y_pred_rus=logreg.predict_proba(X_test)[::,1]
y_pred_rus_cfmatrix=logreg.predict(X_test)
```

[17]:
```
auc_rus = metrics.roc_auc_score(y_test, y_pred_rus)
fpr_rus, tpr_rus, _ = metrics.roc_curve(y_test,  y_pred_rus)
print(auc_rus)
```

```
0.980357178746569
```

[18]:
```
kfold = KFold(n_splits=5, random_state=42, shuffle=True)
model = LogisticRegression(solver='liblinear')
results_rus = cross_val_score(model, X_rus, y_rus, cv=kfold,␣
 ↪scoring="average_precision")
cv_leg_reg_rus_mean = results_rus.mean()*100.0
cv_leg_reg_rus_std = results_rus.std()*100.0

# Output the accuracy. Calculate the mean and std across all folds.
print("Accuracy: %.3f%% (%.3f%%)" % (cv_leg_reg_rus_mean, cv_leg_reg_rus_std))
```

```
Accuracy: 98.233% (0.714%)
```

Q6.1. Perform random over sampling on the development set. What is the shape of your development features? How many positive and negative labels are there in your development set? (Please set random state as 42 when performing random over sampling)

[19]:
```
# Your Code Here
ros = RandomOverSampler(random_state=42)
# resampling X, y
X_ros, y_ros = ros.fit_resample(X_dev, y_dev)
# new class distribution
print(collections.Counter(y_ros))
```

```
Counter({0: 227451, 1: 227451})
```

[20]:
```
print(X_ros.shape)
print(y_ros.shape)
```

```
(454902, 29)
(454902,)
```

4 Answer

definitely the number of samples we have is much larger than we had during under-sampling and it is logical. However, the distribution of 1 and 0 classes is still 50/50 which is still better than original imbalanced dataset

Q6.2. Fit a default logistic regression model using over sampling. Print the AUC and average precision of 5-fold cross validation. (Please set random state as 42 when performing random over sampling)

[21]:
```
# Your Code Here
logreg=LogisticRegression()
logreg.fit(X_ros,y_ros)
y_pred_ros=logreg.predict_proba(X_test)[::,1]
y_pred_ros_cfmatrix=logreg.predict(X_test)
```

[22]:
```
auc_ros = metrics.roc_auc_score(y_test, y_pred_ros)
fpr_ros, tpr_ros, _ = metrics.roc_curve(y_test,  y_pred_ros)
print(auc_ros)
```

```
0.9794992420153205
```

[23]:
```
kfold = KFold(n_splits=5, random_state=42, shuffle=True)
model = LogisticRegression(solver='liblinear')
results_ros = cross_val_score(model, X_ros, y_ros, cv=kfold,␣
 ↪scoring="average_precision")
cv_leg_reg_ros_mean = results_ros.mean()*100.0
cv_leg_reg_ros_std = results_ros.std()*100.0
```

```
# Output the accuracy. Calculate the mean and std across all folds.
print("Accuracy: %.3f%% (%.3f%%)" % (cv_leg_reg_ros_mean, cv_leg_reg_ros_std))
```

```
Accuracy: 98.971% (0.018%)
```

Q7.1. Perform Synthetic Minority Oversampling Technique (SMOTE) on the development set. What is the shape of your development features? How many positive and negative labels are there in your development set? (Please set random state as 42 when performing SMOTE)

[24]:
```
# Your Code Here
oversample = SMOTE(random_state=42)
X_smote, y_smote = oversample.fit_resample(X_dev, y_dev)
```

[25]:
```
print(X_smote.shape)
print(y_smote.shape)
```

```
(454902, 29)
(454902,)
```

[26]:
```
print(collections.Counter(y_smote))
```

```
Counter({0: 227451, 1: 227451})
```

5 Answer

we have the exact same number of negative and positive values however the number of samples is much larger

Q7.2. Fit a default logistic regression model using SMOTE. Print the AUC and average precision of 5-fold cross validation. (Please set random state as 42 when performing SMOTE)

[27]:
```
# Your Code Here
logreg = LogisticRegression()
logreg.fit(X_smote,y_smote)
y_pred_smote = logreg.predict_proba(X_test)[::,1]
y_pred_smote_cfmatrix = logreg.predict(X_test)
y_pred_smote
```

[27]:
```
array([1.        , 0.01536775, 0.0155021 , …, 0.24512392, 0.01504143,
       0.36886868])
```

[28]:
```
auc_smote = metrics.roc_auc_score(y_test, y_pred_smote)
fpr_smote, tpr_smote, _ = metrics.roc_curve(y_test,  y_pred_smote)
print(auc_smote)
```

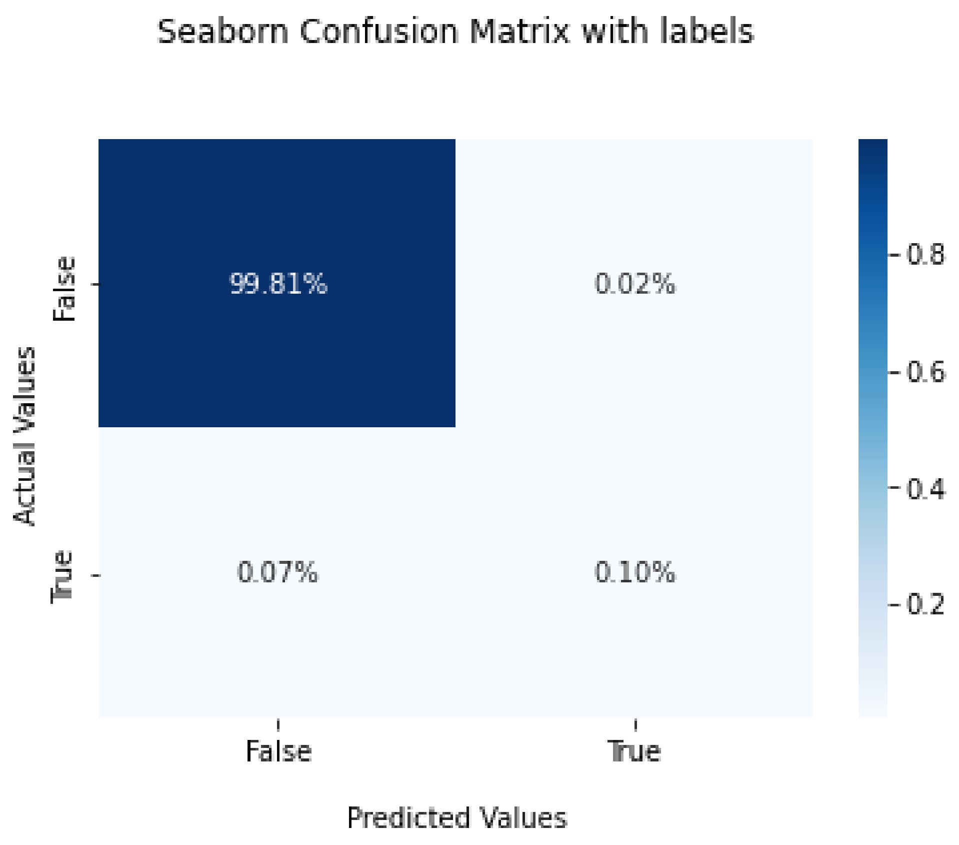
```
0.9790272960619251
```

[29]:
```
kfold = KFold(n_splits=5, random_state=42, shuffle=True)
model = LogisticRegression(solver='liblinear')
results_smote = cross_val_score(model, X_smote, y_smote, cv=kfold,␣
 ↪scoring="average_precision")
cv_leg_reg_smote_mean = results_smote.mean()*100.0
cv_leg_reg_smote_std = results_smote.std()*100.0

# Output the accuracy. Calculate the mean and std across all folds.
print("Accuracy: %.3f%% (%.3f%%)" % (cv_leg_reg_smote_mean,␣
 ↪cv_leg_reg_smote_std))
```

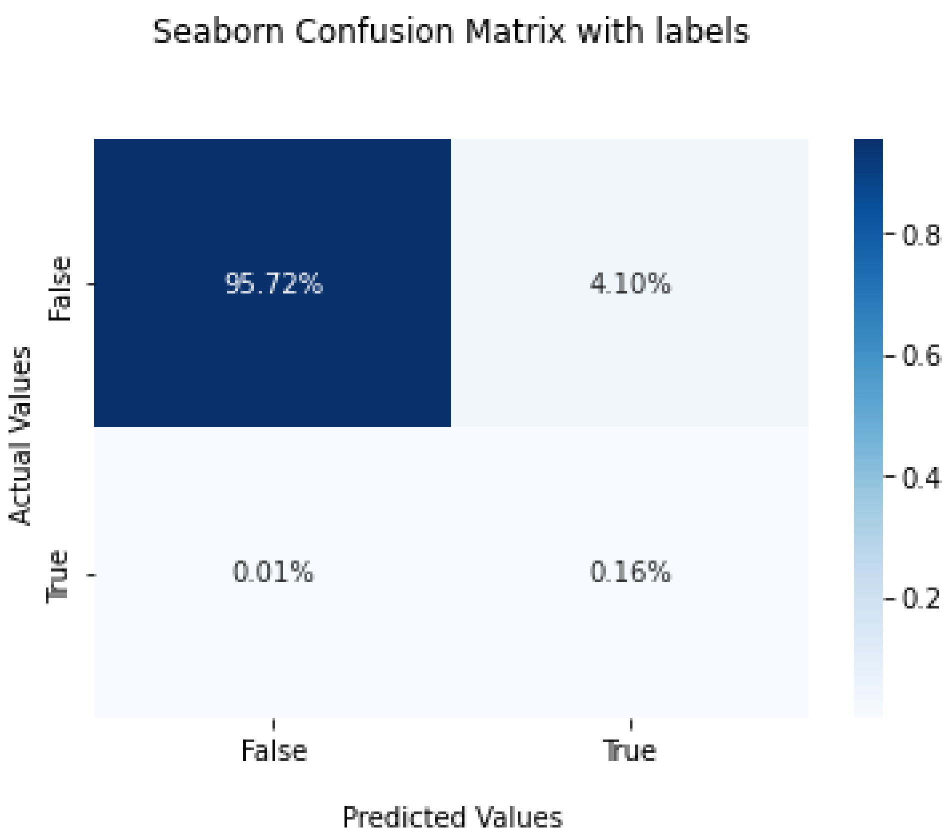
```
Accuracy: 99.121% (0.018%)
```

Q8. Plot confusion matrices on the test set for all four models above. Comment on your result.

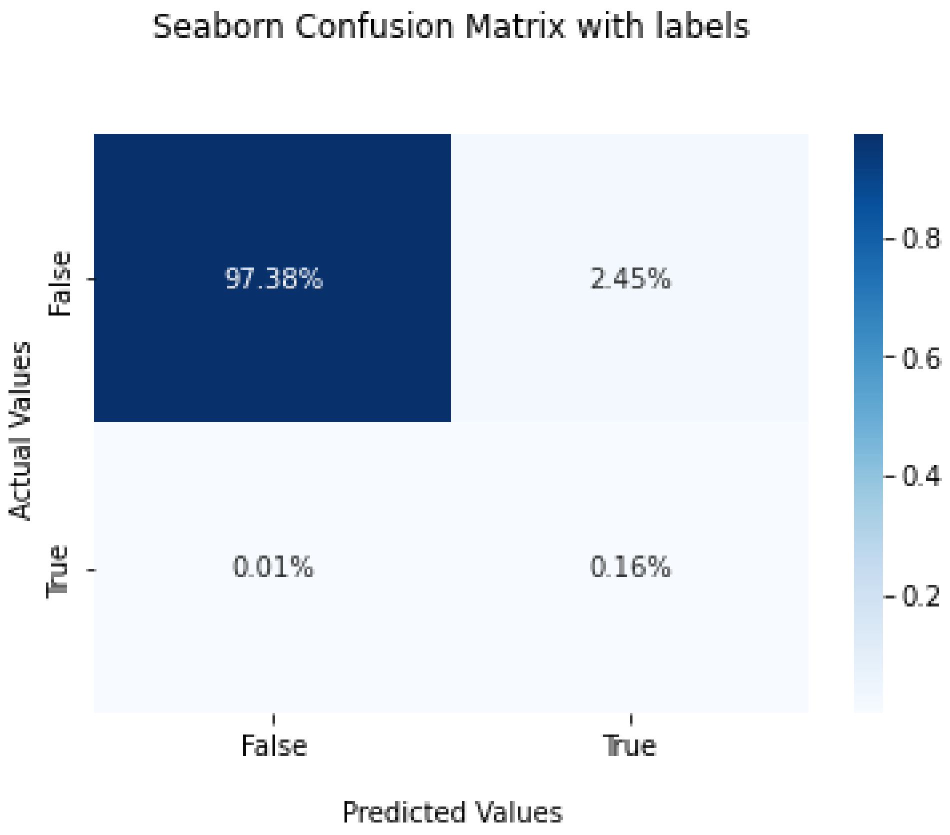
[30]:
```
# Your Code Here
# matrix 1
from sklearn.metrics import confusion_matrix
import seaborn as sns


cf_matrix_1 = confusion_matrix(y_test, y_pred_default_cfmatrix)
ax = sns.heatmap(cf_matrix_1/np.sum(cf_matrix_1),fmt='.2%',  annot=True,␣
 ↪cmap='Blues')

ax.set_title('Seaborn Confusion Matrix with labels\n\n');
ax.set_xlabel('\nPredicted Values')
ax.set_ylabel('Actual Values ');

## Ticket labels - List must be in alphabetical order
ax.xaxis.set_ticklabels(['False','True'])
ax.yaxis.set_ticklabels(['False','True'])

## Display the visualization of the Confusion Matrix.
plt.show()
```



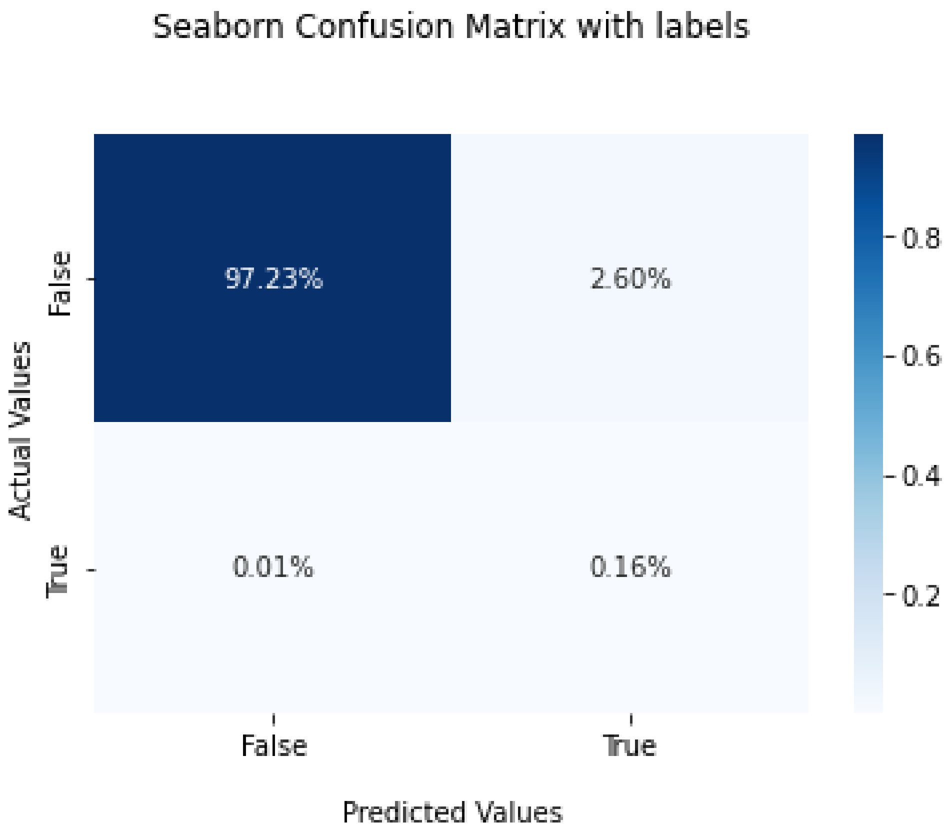
[31]:
```python
# Your Code Here
# matrix 2
from sklearn.metrics import confusion_matrix
import seaborn as sns


cf_matrix_2 = confusion_matrix(y_test, y_pred_rus_cfmatrix)
ax = sns.heatmap(cf_matrix_2/np.sum(cf_matrix_2),fmt='.2%', annot=True,␣
 ↪cmap='Blues')

ax.set_title('Seaborn Confusion Matrix with labels\n\n');
ax.set_xlabel('\nPredicted Values')
ax.set_ylabel('Actual Values ');

## Ticket labels - List must be in alphabetical order
ax.xaxis.set_ticklabels(['False','True'])
ax.yaxis.set_ticklabels(['False','True'])

## Display the visualization of the Confusion Matrix.
plt.show()
```



```
[32]: # Your Code Here
      # matrix 3
      from sklearn.metrics import confusion_matrix
      import seaborn as sns


      cf_matrix_3 = confusion_matrix(y_test, y_pred_ros_cfmatrix)
      ax = sns.heatmap(cf_matrix_3/np.sum(cf_matrix_3),fmt='.2%', annot=True,␣
       ↪cmap='Blues')

      ax.set_title('Seaborn Confusion Matrix with labels\n\n');
      ax.set_xlabel('\nPredicted Values')
      ax.set_ylabel('Actual Values ');

      ## Ticket labels - List must be in alphabetical order
      ax.xaxis.set_ticklabels(['False','True'])
      ax.yaxis.set_ticklabels(['False','True'])

      ## Display the visualization of the Confusion Matrix.
      plt.show()
```

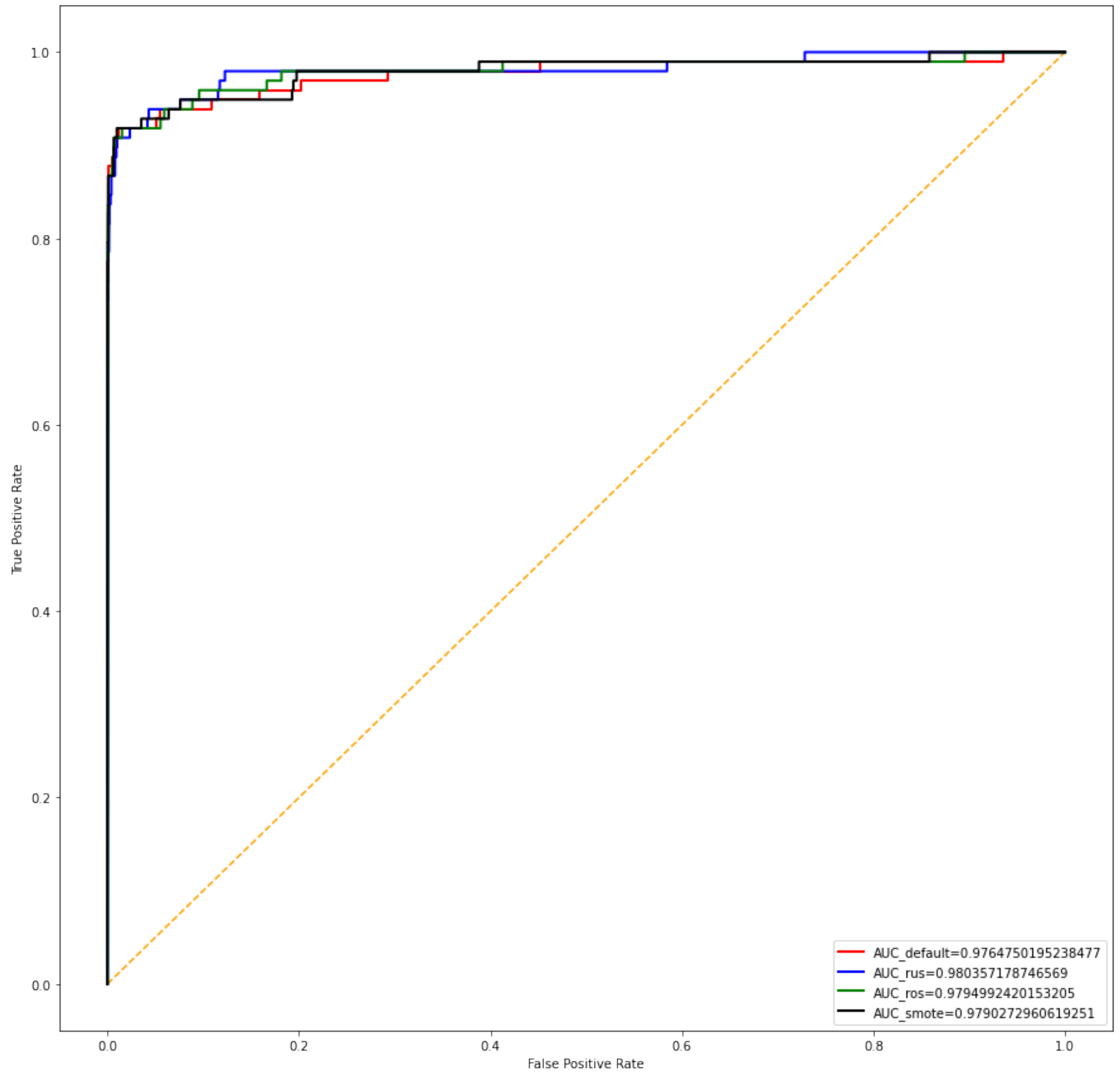


[33]:
```
# Your Code Here
# matrix 4
from sklearn.metrics import confusion_matrix
import seaborn as sns


cf_matrix_4 = confusion_matrix(y_test, y_pred_smote_cfmatrix)
ax = sns.heatmap(cf_matrix_4/np.sum(cf_matrix_4),fmt='.2%', annot=True,␣
 ↪cmap='Blues')

ax.set_title('Seaborn Confusion Matrix with labels\n\n');
ax.set_xlabel('\nPredicted Values')
ax.set_ylabel('Actual Values ');

## Ticket labels - List must be in alphabetical order
ax.xaxis.set_ticklabels(['False','True'])
ax.yaxis.set_ticklabels(['False','True'])

## Display the visualization of the Confusion Matrix.
plt.show()
```



Q9. Plot the ROC for all four models above in a single plot. Make sure to label the axes and legend. Comment on your result.

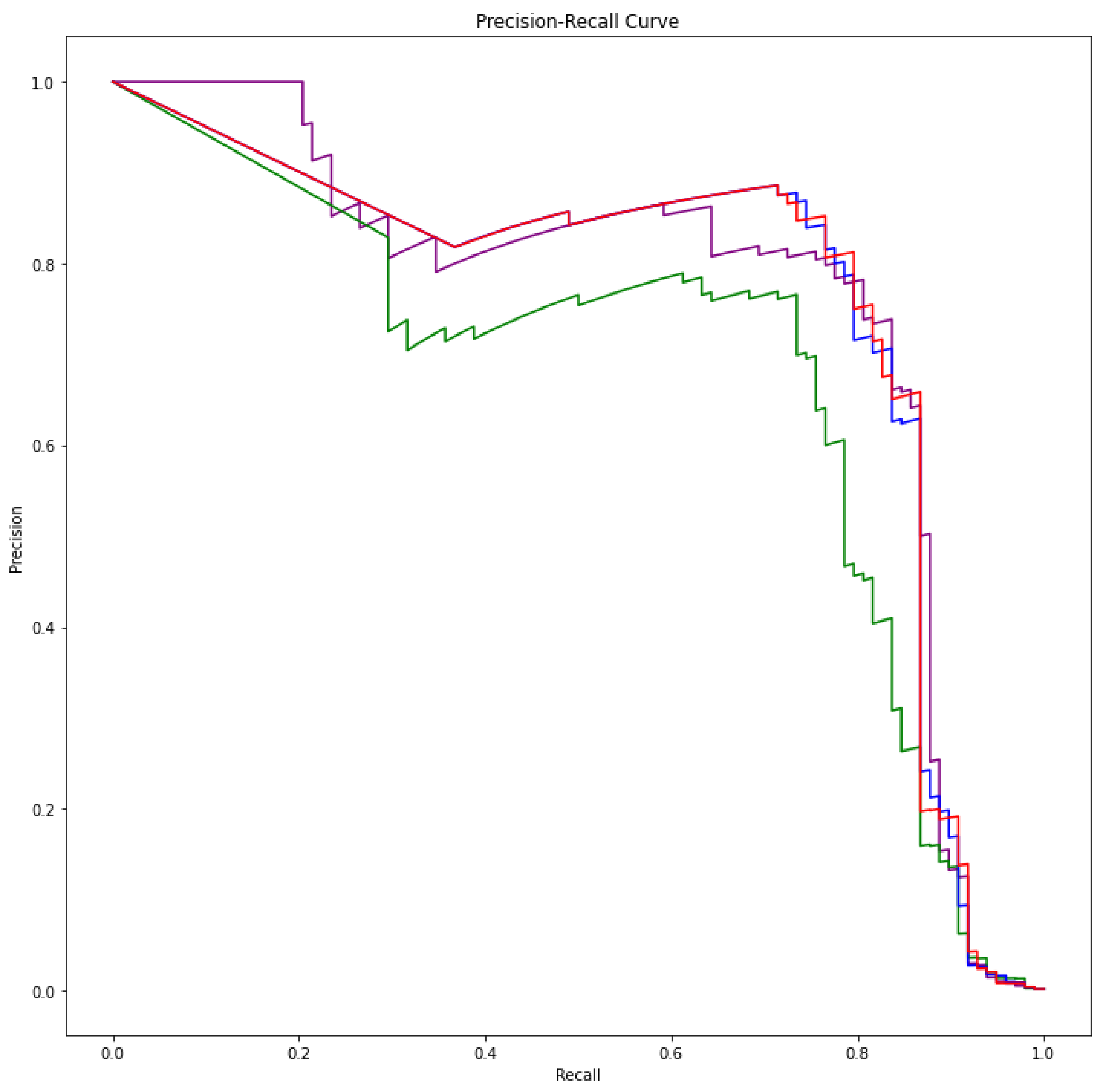
[34]:
```
#create ROC curve
plt.figure(figsize=(15,15),)
plt.plot([0,1], [0,1], color='orange', linestyle='--')
plt.ylabel('True Positive Rate')
plt.xlabel('False Positive Rate')



plt.plot(fpr_default,tpr_default,label="AUC_default="+str(auc_default_legreg),␣
 ↪color='red', linewidth=2)
plt.plot(fpr_rus,tpr_rus,label="AUC_rus="+str(auc_rus), color='blue',␣
 ↪linewidth=2)
plt.plot(fpr_ros,tpr_ros,label="AUC_ros="+str(auc_ros), color='green',␣
 ↪linewidth=2)
plt.plot(fpr_smote,tpr_smote,label="AUC_smote="+str(auc_smote), color='black',␣
 ↪linewidth=2)
plt.legend(loc=4)
plt.show()
```



Q10. Plot the precision-recall curve for all four models above in a single plot. Make sure to label the axes and legend. Comment on your result.

[35]:
```
# Your Code Here
#calculate precision and recall
from sklearn.metrics import precision_recall_curve
from matplotlib.pyplot import figure
```

```
precision_default, recall_default,_ = precision_recall_curve(y_test,␣
 ↪y_pred_default)
precision_rus, recall_rus,_ = precision_recall_curve(y_test, y_pred_rus)
precision_ros, recall_ros,_ = precision_recall_curve(y_test, y_pred_ros)
precision_smote, recall_smote,_ = precision_recall_curve(y_test, y_pred_smote)

#create precision recall curve
fig, ax = plt.subplots(figsize=(12,12))
ax.plot(recall_default, precision_default, color='purple')
ax.plot(recall_rus, precision_rus, color='green')
ax.plot(recall_ros, precision_ros, color='blue')
ax.plot(recall_smote, precision_smote, color='red')

#add axis labels to plot
ax.set_title('Precision-Recall Curve')
ax.set_ylabel('Precision')
ax.set_xlabel('Recall')

#display plot
plt.show()
```



Q11. Adding class weights to a logistic regression model. Print the AUC and average precision of 5-fold cross validation. Also, plot its confusion matrix on test set.

[36]:
```
# Your Code Here

from sklearn.model_selection import GridSearchCV, StratifiedKFold
lr = LogisticRegression(solver='newton-cg')
weights = np.linspace(0.0,0.99,200)

#Creating a dictionary grid for grid search
param_grid = {'class_weight': [{0:x, 1:1.0-x} for x in weights]}

#Fitting grid search to the train data with 5 folds
```

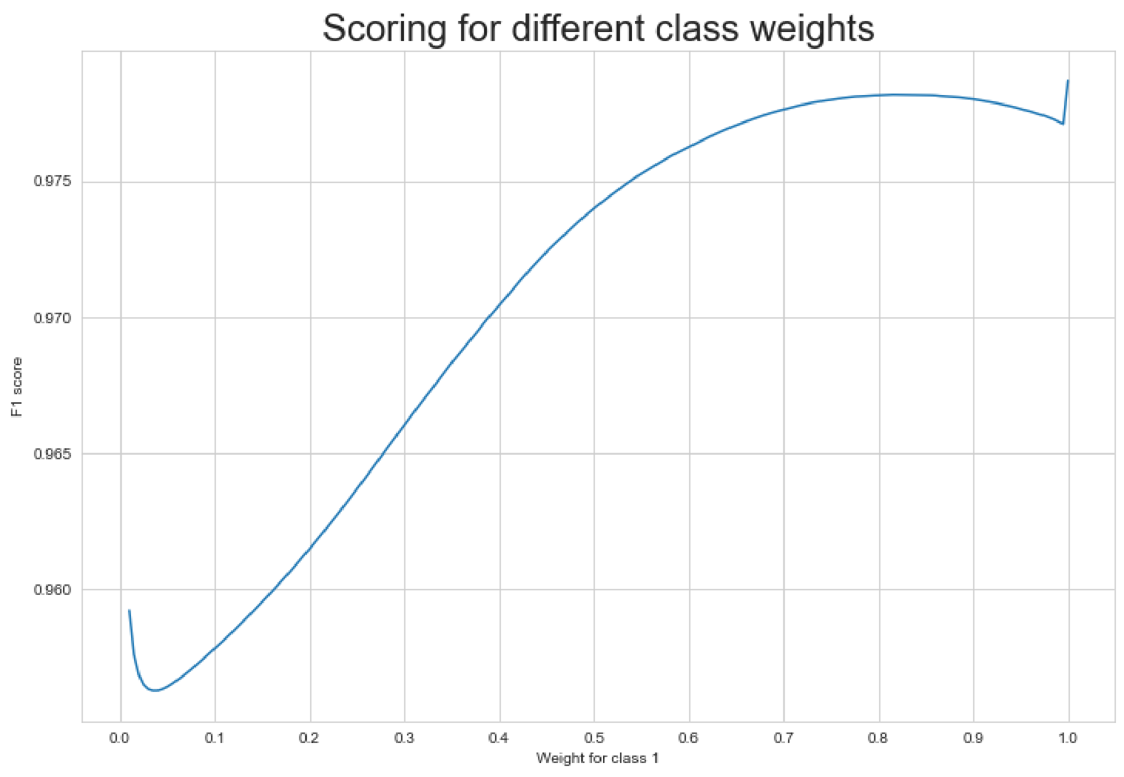
```
gridsearch = GridSearchCV(estimator= lr,
                          param_grid= param_grid,
                          cv=5,
                          n_jobs=-1,
                          scoring='roc_auc',
                          verbose=2).fit(X_dev, y_dev)
```

```
Fitting 5 folds for each of 200 candidates, totalling 1000 fits
```

[37]:
```
#Ploting the score for different values of weight
sns.set_style('whitegrid')
plt.figure(figsize=(12,8))
weigh_data = pd.DataFrame({ 'score': gridsearch.cv_results_['mean_test_score'],␣
 ↪'weight': (1- weights)})
sns.lineplot(weigh_data['weight'], weigh_data['score'])
plt.xlabel('Weight for class 1')
plt.ylabel('F1 score')
plt.xticks([round(i/10,1) for i in range(0,11,1)])
plt.title('Scoring for different class weights', fontsize=24)
```

```
/Users/davitbarblishvili/opt/anaconda3/lib/python3.9/site-
packages/seaborn/_decorators.py:36: FutureWarning: Pass the following variables
as keyword args: x, y. From version 0.12, the only valid positional argument
will be `data`, and passing other arguments without an explicit keyword will
result in an error or misinterpretation.
  warnings.warn(
```

[37]:
```
Text(0.5, 1.0, 'Scoring for different class weights')
```



[38]:
```
#auc_smote = metrics.roc_auc_score(y_test, y_pred_smote)
gridsearch.fit(X_dev,y_dev)
y_pred_balanced=logreg.predict_proba(X_test)[::,1]
y_pred_balanced_cfmatrix=logreg.predict(X_test)
```

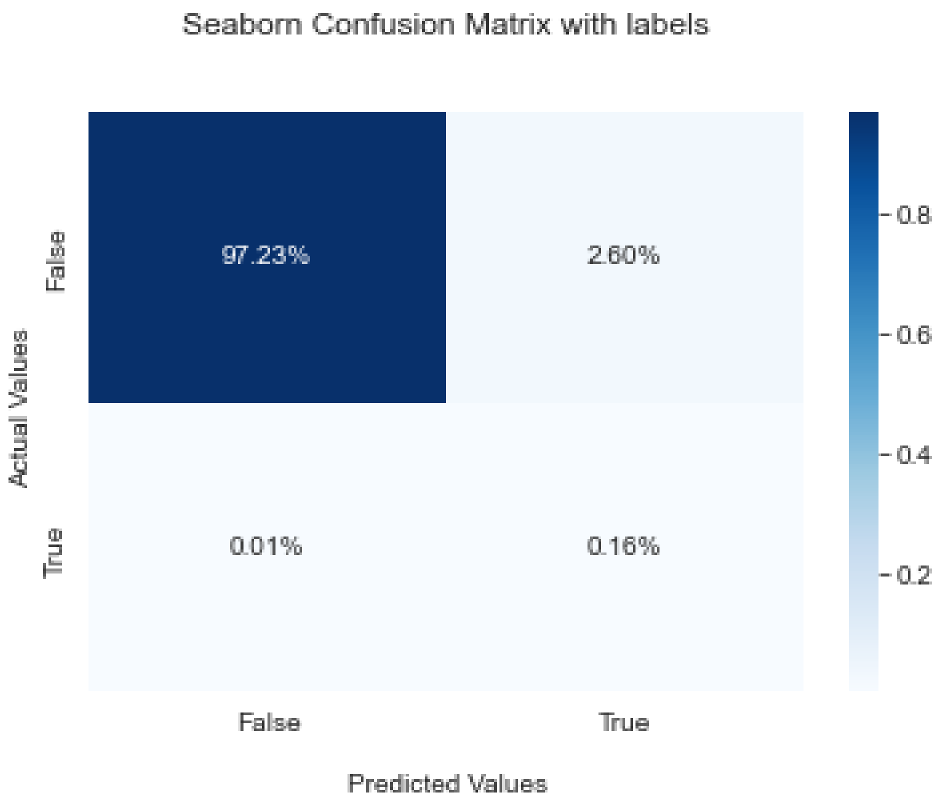
```
Fitting 5 folds for each of 200 candidates, totalling 1000 fits
```

[39]:
```
auc_balanced = metrics.roc_auc_score(y_test, y_pred_balanced)
fpr_balanced, tpr_balanced, _ = metrics.roc_curve(y_test,  y_pred_balanced)
print(auc_balanced)
```

```
0.9790272960619251
```

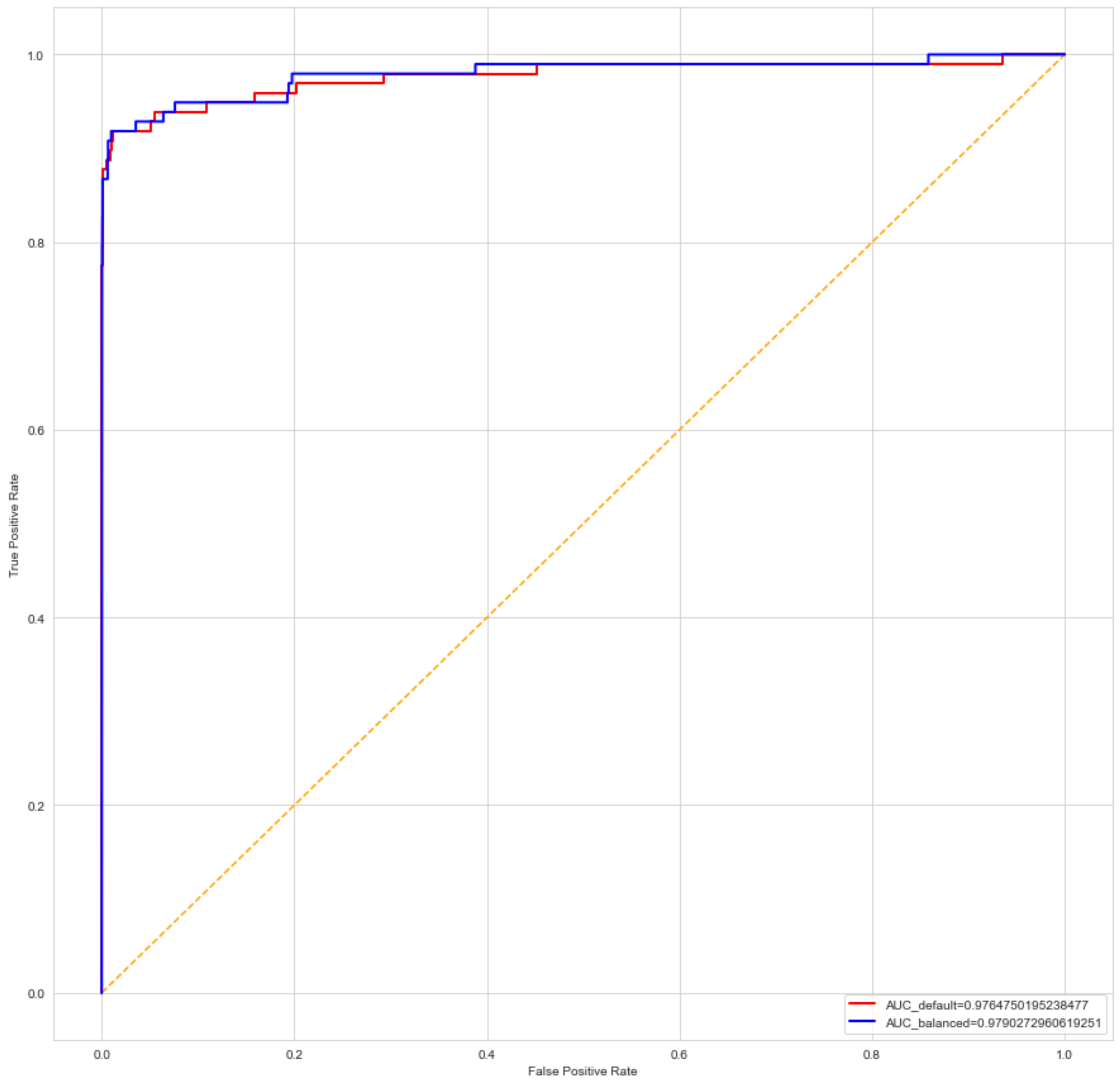
[40]:
```
from sklearn.metrics import confusion_matrix
import seaborn as sns

cf_matrix_balanced = confusion_matrix(y_test, y_pred_balanced_cfmatrix)
ax = sns.heatmap(cf_matrix_balanced/np.sum(cf_matrix_balanced),fmt='.2%',␣
 ↪annot=True, cmap='Blues')

ax.set_title('Seaborn Confusion Matrix with labels\n\n');
ax.set_xlabel('\nPredicted Values')
ax.set_ylabel('Actual Values ');
```

```
## Ticket labels - List must be in alphabetical order
ax.xaxis.set_ticklabels(['False','True'])
ax.yaxis.set_ticklabels(['False','True'])

## Display the visualization of the Confusion Matrix.
plt.show()
```

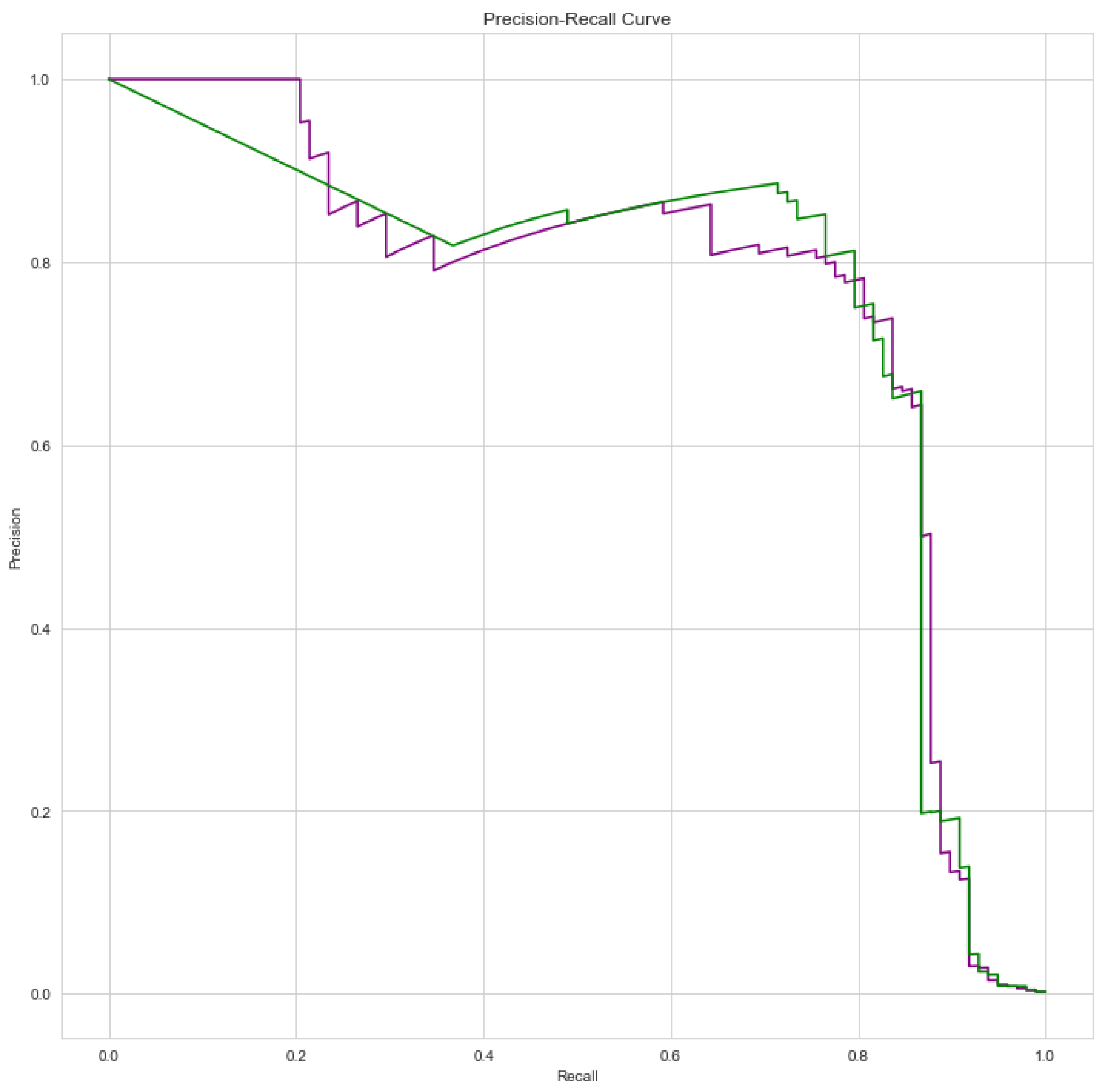


Q12. Plot the ROC and the precision-recall curve for default Logistic without any sampling method and this balanced Logistic model in two single plots. Make sure to label the axes and legend. Comment on your result.

[41]:
```
# Your Code Here
# default logistic
#create ROC curve
plt.figure(figsize=(15,15),)
plt.plot([0,1], [0,1], color='orange', linestyle='--')
plt.ylabel('True Positive Rate')
plt.xlabel('False Positive Rate')
plt.plot(fpr_default,tpr_default,label="AUC_default="+str(auc_default_legreg),␣
↪color='red', linewidth=2)
plt.plot(fpr_balanced,tpr_balanced,label="AUC_balanced="+str(auc_balanced),␣
↪color='blue', linewidth=2)

plt.legend(loc=4)




plt.show()
```



```
[42]: # Your Code Here
      #calculate precision and recall
      from sklearn.metrics import precision_recall_curve
      from matplotlib.pyplot import figure

      precision_default, recall_default,_ = precision_recall_curve(y_test,␣
       ↪y_pred_default)
      precision_balanced, recall_balanced,_ = precision_recall_curve(y_test,␣
       ↪y_pred_balanced)


      #create precision recall curve
      fig, ax = plt.subplots(figsize=(12,12))
      ax.plot(recall_default, precision_default, color='purple')
      ax.plot(recall_balanced, precision_balanced, color='green')

      #add axis labels to plot
      ax.set_title('Precision-Recall Curve')
      ax.set_ylabel('Precision')
      ax.set_xlabel('Recall')

      #display plot
      plt.show()


      # similar results in the end
```



5.1 Part 2: Unsupervised Learning

In this part, we will be applying unsupervised learning approaches to a problem in computational biology. Specifically, we will be analyzing single-cell genomic sequencing data. Single-cell genomics is a set of revolutionary new technologies which can profile the genome of a specimen (tissue, blood, etc.) at the resolution of individual cells. This increased granularity can help capture intercellular heterogeneity, key to better understanding and treating complex genetic diseases such as cancer and Alzheimer's.

Source: 10xgenomics.com/blog/single-cell-rna-seq-an-introductory-overview-and-tools-for-getting-started

A common challenge of genomic datasets is their high-dimensionality: a single observation (a cell, in the case of single-cell data) may have tens of thousands of gene expression features. Fortunately, biology offers a lot of structure - different genes work together in pathways and are co-regulated by gene regulatory networks. Unsupervised learning is widely used to discover this intrinsic structure and prepare the data for further analysis.

5.1.1 Dataset: single-cell RNASeq of mouse brain cells

We will be working with a single-cell RNASeq dataset of mouse brain cells. In the following gene expression matrix, each row represents a cell and each column represents a gene. Each entry in the matrix is a normalized gene expression count - a higher value means that the gene is expressed more in that cell. The dataset has been pre-processed using various quality control and normalization methods for single-cell data.

Data source is on Coursework.

[43]:
```
cell_gene_counts_df = pd.read_csv('data/mouse_brain_cells_gene_counts.csv',␣
 ↪index_col='cell')
cell_gene_counts_df
```

[43]:
```
                         0610005C13Rik  0610007C21Rik  0610007L01Rik  \
cell
A1.B003290.3_38_F.1.1         -0.08093         0.7856          1.334
A1.B003728.3_56_F.1.1         -0.08093        -1.4840         -0.576
A1.MAA000560.3_10_M.1.1       -0.08093         0.6300         -0.576
A1.MAA000564.3_10_M.1.1       -0.08093         0.3809          1.782
A1.MAA000923.3_9_M.1.1        -0.08093         0.5654         -0.576
…                                  …              …             …
E2.MAA000902.3_11_M.1.1       14.98400         1.1550         -0.576
E2.MAA000926.3_9_M.1.1        -0.08093        -1.4840         -0.576
E2.MAA000932.3_11_M.1.1       -0.08093         0.5703         -0.576
E2.MAA000944.3_9_M.1.1        -0.08093         0.3389         -0.576
E2.MAA001894.3_39_F.1.1       -0.08093         0.3816         -0.576

                         0610007N19Rik  0610007P08Rik  0610007P14Rik  \
cell
A1.B003290.3_38_F.1.1          -0.2727        -0.4153        -0.8310
A1.B003728.3_56_F.1.1          -0.2727        -0.4153         1.8350
A1.MAA000560.3_10_M.1.1        -0.2727        -0.4153        -0.2084
A1.MAA000564.3_10_M.1.1        -0.2727        -0.4153         1.0300
A1.MAA000923.3_9_M.1.1         -0.2727        -0.4153        -0.8310
…                                  …              …             …
E2.MAA000902.3_11_M.1.1        -0.2727        -0.4153         0.7530
E2.MAA000926.3_9_M.1.1         -0.2727        -0.4153         1.4720
E2.MAA000932.3_11_M.1.1        -0.2727        -0.4153        -0.8310
E2.MAA000944.3_9_M.1.1         -0.2727        -0.4153        -0.2434
E2.MAA001894.3_39_F.1.1        -0.2727        -0.4153        -0.8310

                         0610007P22Rik  0610009B14Rik  0610009B22Rik  \
cell
A1.B003290.3_38_F.1.1          -0.4692       -0.03146        -0.6035
```

```
A1.B003728.3_56_F.1.1          -0.4692       -0.03146         -0.6035
A1.MAA000560.3_10_M.1.1        -0.4692       -0.03146         -0.6035
A1.MAA000564.3_10_M.1.1        -0.4692       -0.03146          1.2640
A1.MAA000923.3_9_M.1.1         -0.4692       -0.03146         -0.6035
…                                …             …            …
E2.MAA000902.3_11_M.1.1        -0.4692       -0.03146         -0.6035
E2.MAA000926.3_9_M.1.1         -0.4692       -0.03146          1.8120
E2.MAA000932.3_11_M.1.1        -0.4692       -0.03146         -0.6035
E2.MAA000944.3_9_M.1.1         -0.4692       -0.03146         -0.6035
E2.MAA001894.3_39_F.1.1        -0.4692       -0.03146         -0.6035

                         0610009D07Rik  …   Zwint    Zxda    Zxdb    Zxdc  \
cell                                    …
A1.B003290.3_38_F.1.1        -1.021000  … -0.7227 -0.2145 -0.1927 -0.4163
A1.B003728.3_56_F.1.1        -1.021000  … -0.7227 -0.2145 -0.1927 -0.4163
A1.MAA000560.3_10_M.1.1       1.253000  …  1.3150 -0.2145 -0.1927 -0.4163
A1.MAA000564.3_10_M.1.1      -1.021000  … -0.7227 -0.2145 -0.1927 -0.4163
A1.MAA000923.3_9_M.1.1       -1.021000  … -0.7227 -0.2145 -0.1927 -0.4163
…                                  …  …     …      …      …      …
E2.MAA000902.3_11_M.1.1      -1.021000  …  1.4260 -0.2145 -0.1927 -0.4163
E2.MAA000926.3_9_M.1.1        1.079000  … -0.7227 -0.2145 -0.1927 -0.4163
E2.MAA000932.3_11_M.1.1      -0.003473  … -0.7227 -0.2145 -0.1927 -0.4163
E2.MAA000944.3_9_M.1.1        1.281000  …  1.2160 -0.2145 -0.1927 -0.4163
E2.MAA001894.3_39_F.1.1       1.106000  … -0.7227 -0.2145 -0.1927 -0.4163

                         Zyg11b     Zyx  Zzef1    Zzz3        a  l7Rn6
cell
A1.B003290.3_38_F.1.1   -0.5923 -0.5913 -0.553 -0.5654 -0.04385  1.567
A1.B003728.3_56_F.1.1   -0.5923 -0.5913 -0.553 -0.5654 -0.04385 -0.681
A1.MAA000560.3_10_M.1.1 -0.5923 -0.5913  2.072 -0.5654 -0.04385  1.260
A1.MAA000564.3_10_M.1.1 -0.5923  2.3900 -0.553  0.1697 -0.04385 -0.681
A1.MAA000923.3_9_M.1.1   2.3180 -0.5913 -0.553 -0.5654 -0.04385 -0.681
…                           …      …     …      …       …     …
E2.MAA000902.3_11_M.1.1 -0.5923 -0.5913 -0.553 -0.5654 -0.04385  1.728
E2.MAA000926.3_9_M.1.1   0.2422 -0.5913 -0.553  1.6060 -0.04385 -0.681
E2.MAA000932.3_11_M.1.1 -0.5923 -0.5913 -0.553 -0.5654 -0.04385  2.074
E2.MAA000944.3_9_M.1.1  -0.5923 -0.5913 -0.553  1.3070 -0.04385 -0.681
E2.MAA001894.3_39_F.1.1 -0.5923 -0.5913 -0.553 -0.5654 -0.04385  1.628

[1000 rows x 18585 columns]
```

Note the dimensionality - we have 1000 cells (observations) and 18,585 genes (features)!

We are also provided a metadata file with annotations for each cell (e.g. cell type, subtissue, mouse sex, etc.)

```
[44]: cell_metadata_df = pd.read_csv('data/mouse_brain_cells_metadata.csv')
      cell_metadata_df
```

```
[44]:                        cell cell_ontology_class    subtissue mouse.sex  \
      0      A1.B003290.3_38_F.1.1           astrocyte     Striatum         F
      1      A1.B003728.3_56_F.1.1           astrocyte     Striatum         F
      2    A1.MAA000560.3_10_M.1.1     oligodendrocyte       Cortex         M
      3    A1.MAA000564.3_10_M.1.1    endothelial cell     Striatum         M
      4     A1.MAA000923.3_9_M.1.1           astrocyte  Hippocampus         M
      ..                       …                   …          …       …
      995  E2.MAA000902.3_11_M.1.1           astrocyte     Striatum         M
      996   E2.MAA000926.3_9_M.1.1     oligodendrocyte       Cortex         M
      997  E2.MAA000932.3_11_M.1.1    endothelial cell  Hippocampus         M
      998   E2.MAA000944.3_9_M.1.1     oligodendrocyte       Cortex         M
      999  E2.MAA001894.3_39_F.1.1     oligodendrocyte       Cortex         F

          mouse.id plate.barcode  n_genes   n_counts
      0     3_38_F       B003290     3359   390075.0
      1     3_56_F       B003728     1718   776436.0
      2     3_10_M     MAA000560     3910  1616084.0
      3     3_10_M     MAA000564     4352   360004.0
      4      3_9_M     MAA000923     2248   290282.0
      ..       …             …      …        …
      995   3_11_M     MAA000902     3026  3134463.0
      996    3_9_M     MAA000926     3085   744301.0
      997   3_11_M     MAA000932     2277   519257.0
      998    3_9_M     MAA000944     3234  1437895.0
      999   3_39_F     MAA001894     3375   885166.0

      [1000 rows x 8 columns]
```

Different cell types

```
[45]: cell_metadata_df['cell_ontology_class'].value_counts()
```

```
[45]: oligodendrocyte                  385
      endothelial cell                 264
      astrocyte                        135
      neuron                            94
      brain pericyte                    58
      oligodendrocyte precursor cell    54
      Bergmann glial cell               10
      Name: cell_ontology_class, dtype: int64
```

Different subtissue types (parts of the brain)

```
[46]: cell_metadata_df['subtissue'].value_counts()
```

```
[46]: Cortex         364
      Hippocampus    273
      Striatum       220
```

```
Cerebellum     143
Name: subtissue, dtype: int64
```

Our goal in this exercise is to use dimensionality reduction and clustering to visualize and better understand the high-dimensional gene expression matrix. We will use the following pipeline, which is common in single-cell analysis: 1. Use PCA to project the gene expression matrix to a lower-dimensional linear subspace. 2. Cluster the data using K-means on the first 20 principal components. 3. Use t-SNE to project the first 20 principal components onto two dimensions. Visualize the points and color by their clusters from (2).

5.2 1 PCA

Q1. Perform PCA and project the gene expression matrix onto its first 50 principal components. You may use `sklearn.decomposition.PCA`.

[47]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
%matplotlib inline
from sklearn.decomposition import PCA
from sklearn.manifold import TSNE
```

[48]:
```
### Your code here
pca = PCA(n_components=50)
principalComponents = pca.fit_transform(cell_gene_counts_df)
principalDf = pd.DataFrame(data = principalComponents)
principalDf.head()
```

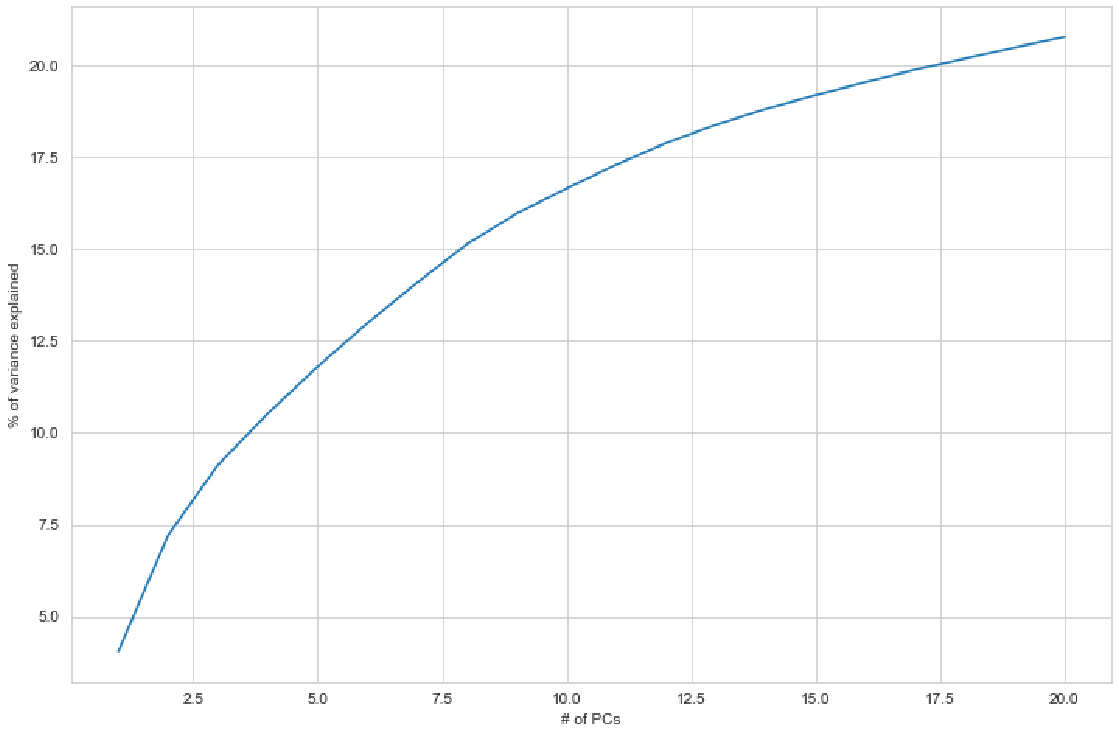
[48]:
```
          0          1          2          3          4          5   \
0  15.353967  22.551441  28.909568  18.160747 -63.669865  63.397356
1 -19.092789  -3.011189  37.073015  -7.781964  -0.324305  -5.520998
2   1.624026 -26.093832  -8.735882   1.431624   3.908803  -0.872088
3 -15.469770  37.906454 -37.408305   5.952024 -10.229875   4.293257
4 -15.223271  -2.999145  38.531674  -6.379690  -6.113621  -4.637017

          6           7         8          9   …        40        41  \
0  22.120360  193.168109  5.079544 -12.085422  …  4.401962 -4.204018
1   1.450258   -0.053582 -2.177476   3.883114  … -0.960823  0.739859
2  -2.047054    2.420198  3.514791   3.970593  … -0.176208  0.610638
3  15.286247   -4.262447 -6.748037   6.366079  … -0.170163 -0.878926
4   5.044908   -2.089758 -6.841563   3.252708  …  0.661995 -1.800574

          42        43         44        45         46        47        48  \
0 -16.134101 -4.409253 -10.119978  1.414017 -18.772922  3.200578 -1.485832
1  -0.799829  0.181566  -0.265277  1.278628  -0.226157 -1.436482 -0.227928
2   0.400863 -0.646152  -0.619592 -0.238504  -0.644655 -0.074413  0.776929
3   2.428335  6.114298   0.626601 -4.534436  -4.646113 -5.116957 -1.471581
4   0.278585  1.744370   0.849674  0.743262   2.260802 -2.302700 -0.416203
```

```
          49
0 -6.014058
1 -1.708477
2  0.975347
3  0.919587
4  0.074911

[5 rows x 50 columns]
```

Q2. Plot the cumulative proportion of variance explained as a function of the number of principal components. How much of the total variance in the dataset is explained by the first 20 principal components?

[49]:
```
### Your code here

fig = plt.figure(figsize=(12,8))
top_20_pca_var = pca.explained_variance_ratio_[:20]
ax = fig.add_subplot(1,1,1)
plt.plot(np.arange(1,21), top_20_pca_var.cumsum()*100)
ax.set_xlabel("# of PCs")
ax.set_ylabel("% of variance explained")
```

[49]:
```
Text(0, 0.5, '% of variance explained')
```



Q3. For the first principal component, report the top 10 loadings (weights) and their corresponding gene names. In other words, which 10 genes are weighted the most in the first principal component?

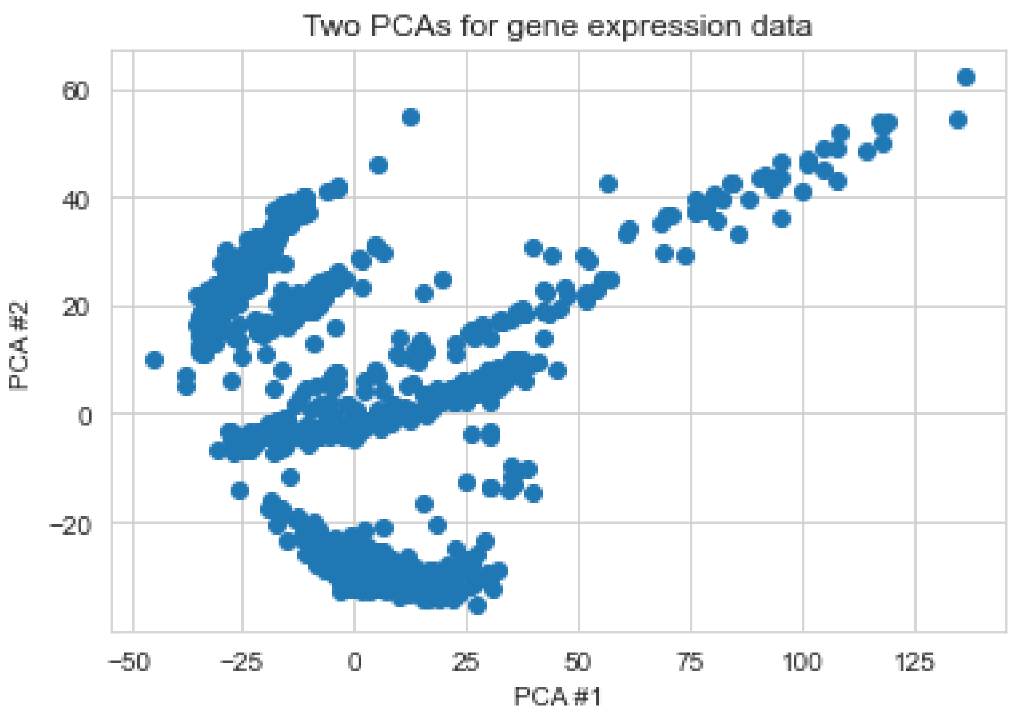
[50]:
```
### Your code here
weights = pca.components_
first_component = abs(weights[0])
ind = np.argpartition(first_component, -10)[-10:]

col_names = cell_gene_counts_df.columns[ind]
col_names
print(col_names)
```

```
Index(['Erc2', 'Cpne5', 'Hpca', 'Nrsn2', 'Camkv', 'Nsg2', 'Rasgef1a', 'Kcnj4',
       'Ptpn5', 'St8sia3'],
      dtype='object')
```

Q4. Plot the projection of the data onto the first two principal components using a scatter plot.

[51]:
```
### Your code here
plt.scatter(x=principalDf[0], y=principalDf[1])
plt.xlabel("PCA #1")
plt.ylabel("PCA #2")
plt.title('Two PCAs for gene expression data')
```

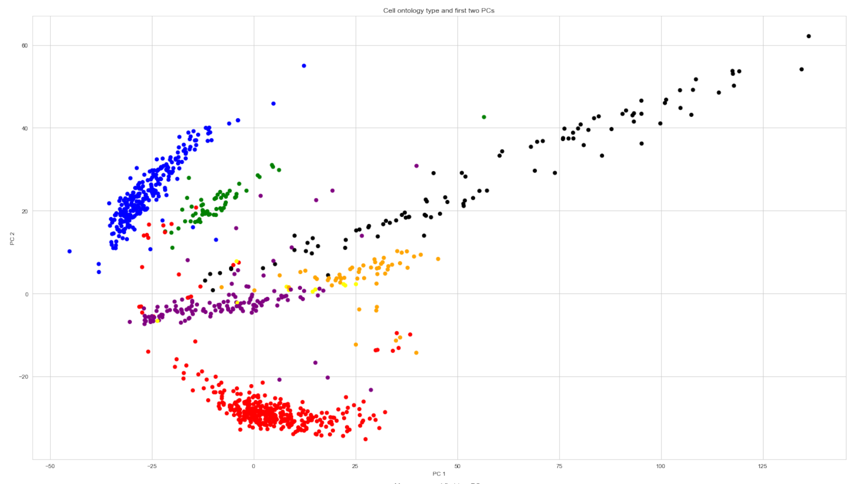
[51]:
```
Text(0.5, 1.0, 'Two PCAs for gene expression data')
```

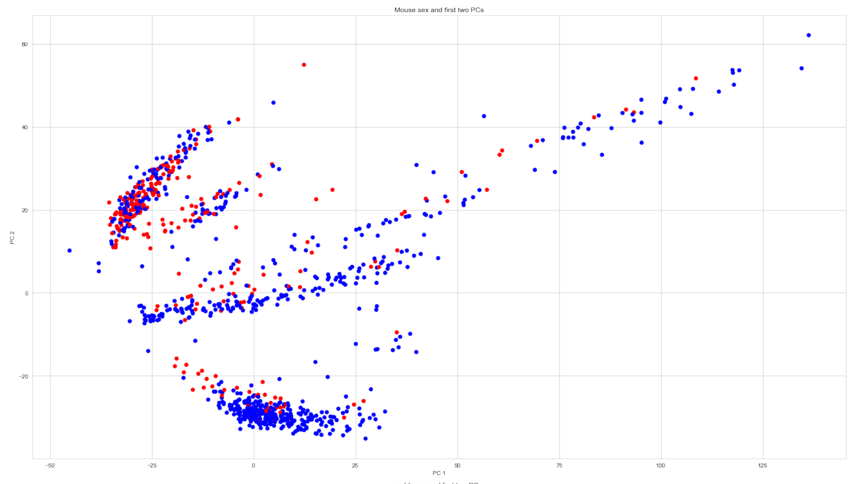


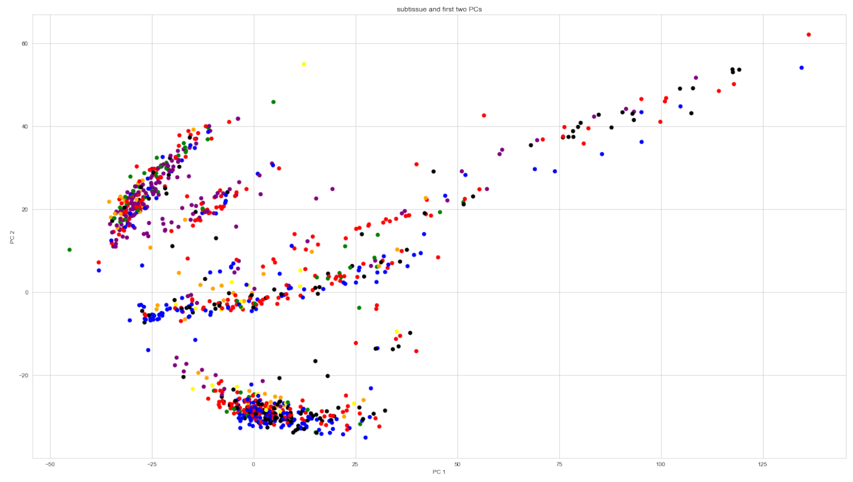
Q5. Now, use a small multiple of four scatter plots to make the same plot as above, but colored by four annotations in the metadata: cell_ontology_class, subtissue, mouse.sex, mouse.id. Include a legend for the labels. For example, one of the plots should have points projected onto PC 1 and PC 2, colored by their cell_ontology_class.

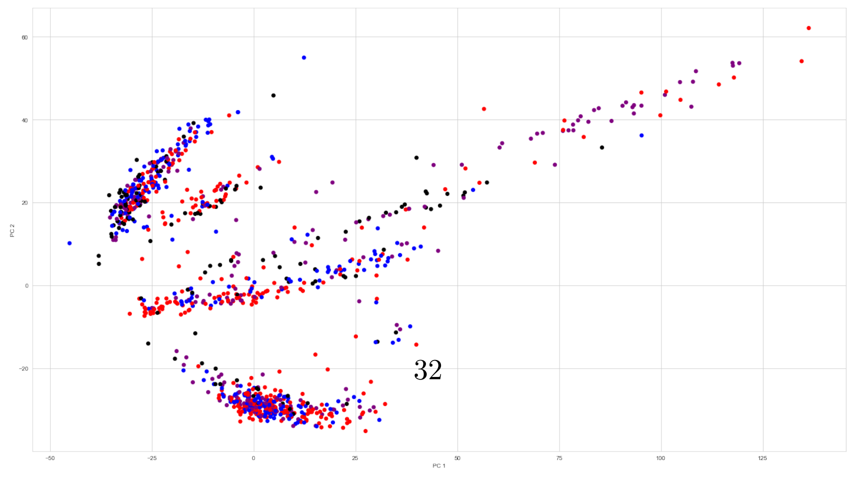
[52]:
```python
### Your code here
fig, (ax1, ax2, ax3, ax4) = plt.subplots(4, figsize=(20,45))
cell_ontology_colors = ['red', 'blue', 'yellow', 'purple', 'black', 'green',␣
 ↪'orange']
cell_ontology_colors_map = {
  "oligodendrocyte": "red",
  "endothelial cell": "blue",
  "astrocyte": "purple",
  "neuron": "black",
  "brain pericyte": "green",
  "oligodendrocyte precursor cell": "orange",
  "Bergmann glial cell": "yellow",
}

mouse_sex_colors_map = {
  "F": "red",
  "M": "blue"
}

mouse_id_colors_map = {
  "3_10_M": "red",
  "3_9_M": "blue",
  "3_38_F": "purple",
  "3_8_M": "black",
  "3_11_M": "green",
  "3_39_F": "orange",
  "3_56_F": "yellow",
}

cell_subtissue_colors_map = {
  "Cortex": "red",
  "Hippocampus": "blue",
  "Striatum": "purple",
  "Cerebellum": "black",
}

cell_metadata_df['ontology_color'] = cell_metadata_df.apply(lambda row :␣
 ↪cell_ontology_colors_map[row['cell_ontology_class']],axis=1)
cell_metadata_df['sex_color'] = cell_metadata_df.apply(lambda row :␣
 ↪mouse_sex_colors_map[row['mouse.sex']],axis=1)
```

```
cell_metadata_df['id_color'] = cell_metadata_df.apply(lambda row :␣
↪mouse_id_colors_map[row['mouse.id']],axis=1)
cell_metadata_df['subtissue_color'] = cell_metadata_df.apply(lambda row :␣
↪cell_subtissue_colors_map[row['subtissue']],axis=1)

ax1.scatter(x=principalDf[0], y=principalDf[1],␣
↪c=cell_metadata_df['ontology_color'])
ax1.set_xlabel("PC 1")
ax1.set_ylabel("PC 2")
ax1.set_title("Cell ontology type and first two PCs")
ax2.scatter(x=principalDf[0], y=principalDf[1], c=cell_metadata_df['sex_color'])
ax2.set_xlabel("PC 1")
ax2.set_ylabel("PC 2")
ax2.set_title("Mouse.sex and first two PCs")
ax3.scatter(x=principalDf[0], y=principalDf[1], c=cell_metadata_df['id_color'])
ax3.set_xlabel("PC 1")
ax3.set_ylabel("PC 2")
ax3.set_title("Mouse.id and first two PCs")
ax4.scatter(x=principalDf[0], y=principalDf[1],␣
↪c=cell_metadata_df['subtissue_color'])
ax4.set_xlabel("PC 1")
ax4.set_ylabel("PC 2")
ax3.set_title("subtissue and first two PCs")


fig.tight_layout()
```


Cell ontology type and first two PCs
PC 1
PC 2
60
40
20
0
−20
−50
−25
0
25
50
75
100
125


Mouse sex and first two PCs
PC 1
PC 2
60
40
20
0
−20
−50
−25
0
25
50
75
100
125


subtissue and first two PCs
PC 1
PC 2
60
40
20
0
−20
−50
−25
0
25
50
75
100
125


PC 1
PC 2
60
40
20
0
−20
−50
−25
0
25
50
75
100
125

Q6. Based on the plots above, the first two principal components correspond to which aspect of the cells? What is the intrinsic dimension that they are describing?

5.2.1 Your answer here

PC1 and PC2 are able to distinguish well between cell ontology type and M vs Female best, so I think that these two components correpond to these aspects of the cells

5.3 Part 2: K-means

While the annotations provide high-level information on cell type (e.g. cell_ontology_class has 7 categories), we may also be interested in finding more granular subtypes of cells. To achieve this, we will use K-means clustering to find a large number of clusters in the gene expression dataset. Note that the original gene expression matrix had over 18,000 noisy features, which is not ideal for clustering. So, we will perform K-means clustering on the first 20 principal components of the dataset.

Q7. Implement a `kmeans` function which takes in a dataset `X` and a number of clusters `k`, and returns the cluster assignment for each point in `X`. You may NOT use sklearn for this implementation. Use lecture 6, slide 14 as a reference.

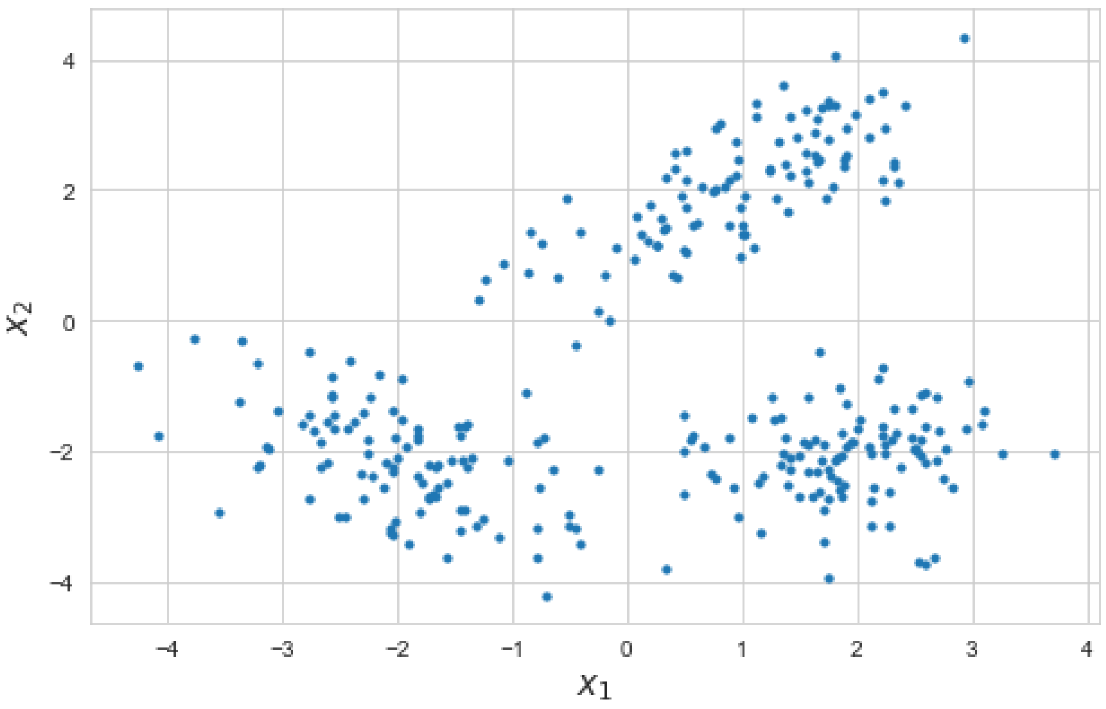
[53]:
```
import random
from scipy.spatial import distance
```

[54]:
```
def kmeans(X, k, iters=10):
    '''Groups the points in X into k clusters using the K-means algorithm.

    Parameters
    ----------
    X : (m x n) data matrix
    k: number of clusters
    iters: number of iterations to run k-means loop

    Returns
    -------
    y: (m x 1) cluster assignment for each point in X
    '''
    ### Your code here
    count = 0
    m = len(X)
    idx = np.random.choice(m, k, replace=False)
    n = len(X[0])
    centroids = X[idx, :]
    distances = distance.cdist(X, centroids, 'euclidean')
    min_ks = np.array([np.argmin(i) for i in distances])
    while count < iters:
        centroids = []
        for idx in range(k):
            centroids.append(X[min_ks==idx].mean(axis=0))

        centroids = np.vstack(centroids)
        distances = distance.cdist(X, centroids ,'euclidean')
        min_ks = np.array([np.argmin(i) for i in distances])

        count = count +1

    return min_ks
```

Before applying K-means on the gene expression data, we will test it on the following synthetic dataset to make sure that the implementation is working.

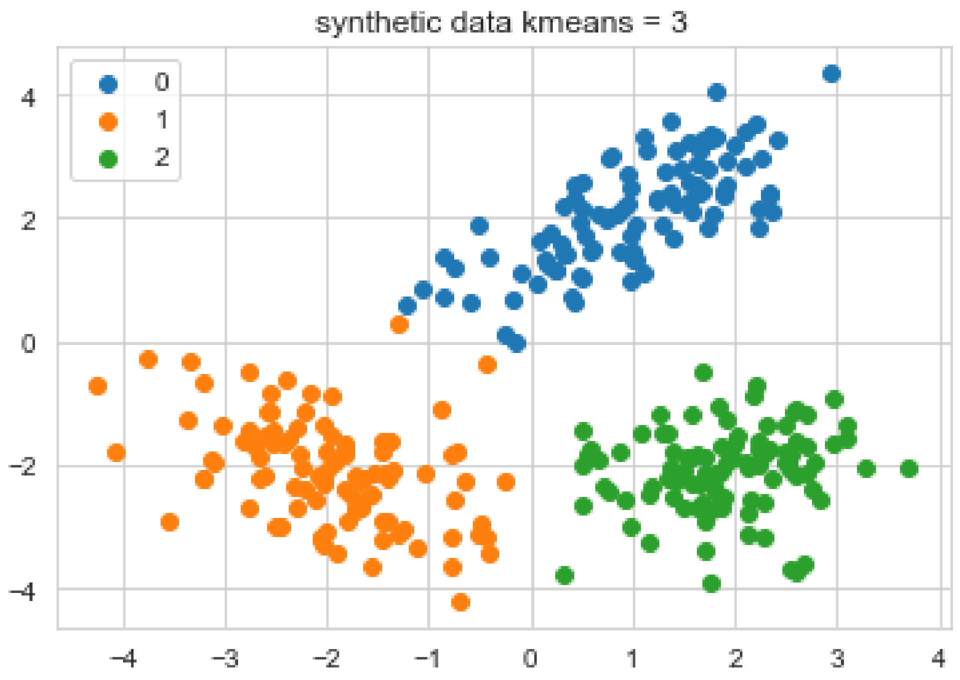
```
[55]: np.random.seed(0)
x_1 = np.random.multivariate_normal(mean=[1, 2], cov=np.array([[0.8, 0.6], [0.
 ↪6, 0.8]]), size=100)
x_2 = np.random.multivariate_normal(mean=[-2, -2], cov=np.array([[0.8, -0.4],␣
 ↪[-0.4, 0.8]]), size=100)
x_3 = np.random.multivariate_normal(mean=[2, -2], cov=np.array([[0.4, 0], [0, 0.
 ↪4]]), size=100)
X = np.vstack([x_1, x_2, x_3])

plt.figure(figsize=(8, 5))
plt.scatter(X[:, 0], X[:, 1], s=10)
plt.xlabel('$x_1$', fontsize=15)
plt.ylabel('$x_2$', fontsize=15)
```

```
[55]: Text(0, 0.5, '$x_2$')
```



Q8. Apply K-means with k=3 to the synthetic dataset above. Plot the points colored by their K-means cluster assignments to verify that your implementation is working.

[56]:
```
### Your code here
label = kmeans(X, 3, 25)

for i in np.unique(label):
    plt.scatter(X[label == i , 0] , X[label == i , 1] , label = i)
plt.legend()
plt.title('synthetic data kmeans = 3')
plt.show()
```



Q9. Use K-means with k=20 to cluster the first 20 principal components of the gene expression data.

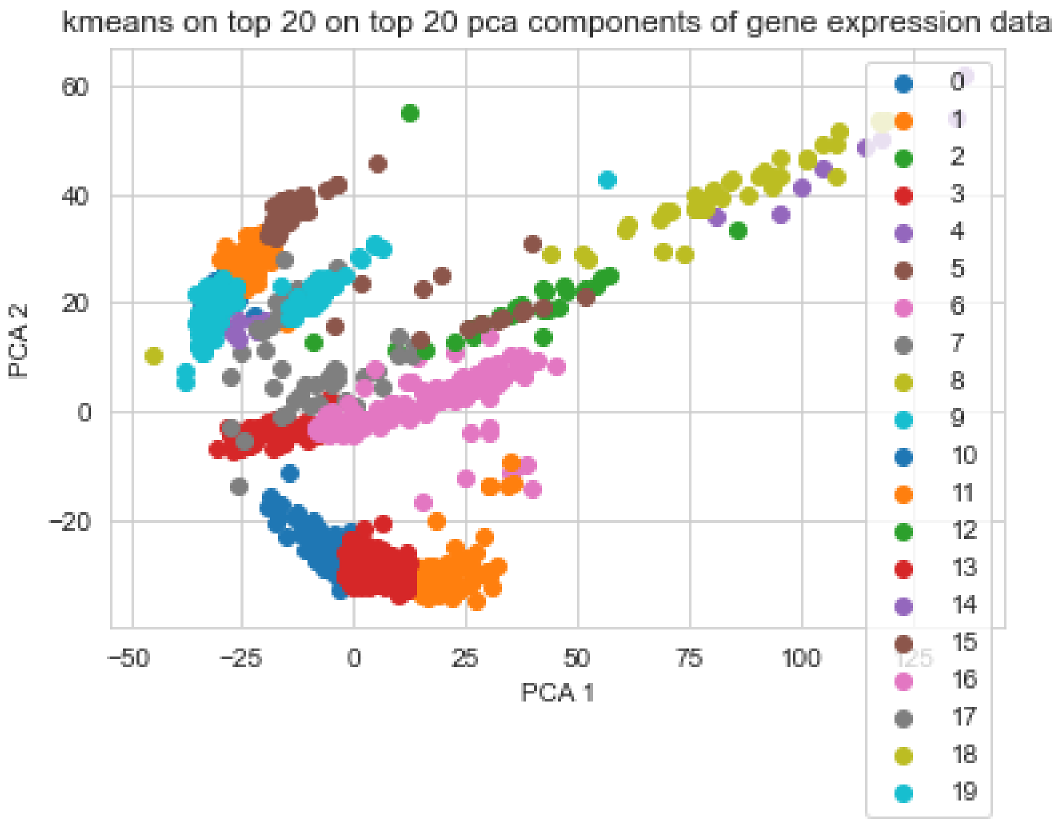
```
[57]: ### Your code here
      pca = PCA(n_components=20)
      principalComponents = pca.fit_transform(cell_gene_counts_df)
      principalDf_20 = pd.DataFrame(data = principalComponents)

      principalDf_20.head()
```

```
[57]:            0          1          2          3          4          5   \
      0  15.353967  22.551441  28.909567  18.160721 -63.669732  63.397438
      1 -19.092789  -3.011189  37.073015  -7.781967  -0.324282  -5.521006
      2   1.624026 -26.093832  -8.735882   1.431620   3.908809  -0.872080
      3 -15.469770  37.906454 -37.408306   5.952010 -10.229885   4.293242
      4 -15.223271  -2.999145  38.531674  -6.379688  -6.113613  -4.637026

                6           7         8          9         10         11         12  \
      0  22.120252  193.167578  5.080296 -12.099356 -6.763981 -10.413467 -3.761505
      1   1.450235   -0.053612 -2.177560   3.883312  3.890762  -1.156763 -5.808779
      2  -2.047059    2.420158  3.514868   3.969962  0.178760  -0.663556 -4.586811
      3  15.286236   -4.262500 -6.747644   6.366951 -0.890539   3.889235 -1.977652
      4   5.044897   -2.089760 -6.841607   3.252973  6.327452   4.271913  2.166967

               13         14         15         16         17         18          19
```

```
0  7.433519  29.834380 -82.694701 -34.188857  56.933983  18.914266 -133.453789
1  2.198223   6.359785   1.536259   3.743419   0.811467   0.743786   -0.172512
2  0.337201   2.795735   0.164959   3.340482  -0.914410  -0.523098   -0.243974
3  6.868857   4.294097  -0.169746  -0.760807  -3.756193   0.962947   -1.504033
4 -1.048801  -0.788229   0.269816  -1.451772   0.635045  -0.025509   -1.123405
```

[58]:
```
principalDf_20_numpy = principalDf_20.to_numpy()

pca_labels = kmeans(principalDf_20_numpy, 20, 15)
```

[59]:
```
for i in np.unique(pca_labels):
    plt.scatter(principalDf_20_numpy[pca_labels == i , 0] ,␣
 ↪principalDf_20_numpy[pca_labels == i , 1] , label = i)

plt.xlabel('PCA 1')
plt.ylabel('PCA 2')
plt.title('kmeans on top 20 on top 20 pca components of gene expression data')
plt.legend()
plt.show()
```



5.4 3 t-SNE

In this final section, we will visualize the data again using t-SNE - a non-linear dimensionality reduction algorithm. You can learn more about t-SNE in this interactive tutorial: https://distill.pub/2016/misread-tsne/.

Q10. Use t-SNE to reduce the first 20 principal components of the gene expression dataset to two dimensions. You may use `sklearn.manifold.TSNE`. Note that it is recommended to first perform PCA before applying t-SNE to suppress noise and speed up computation.

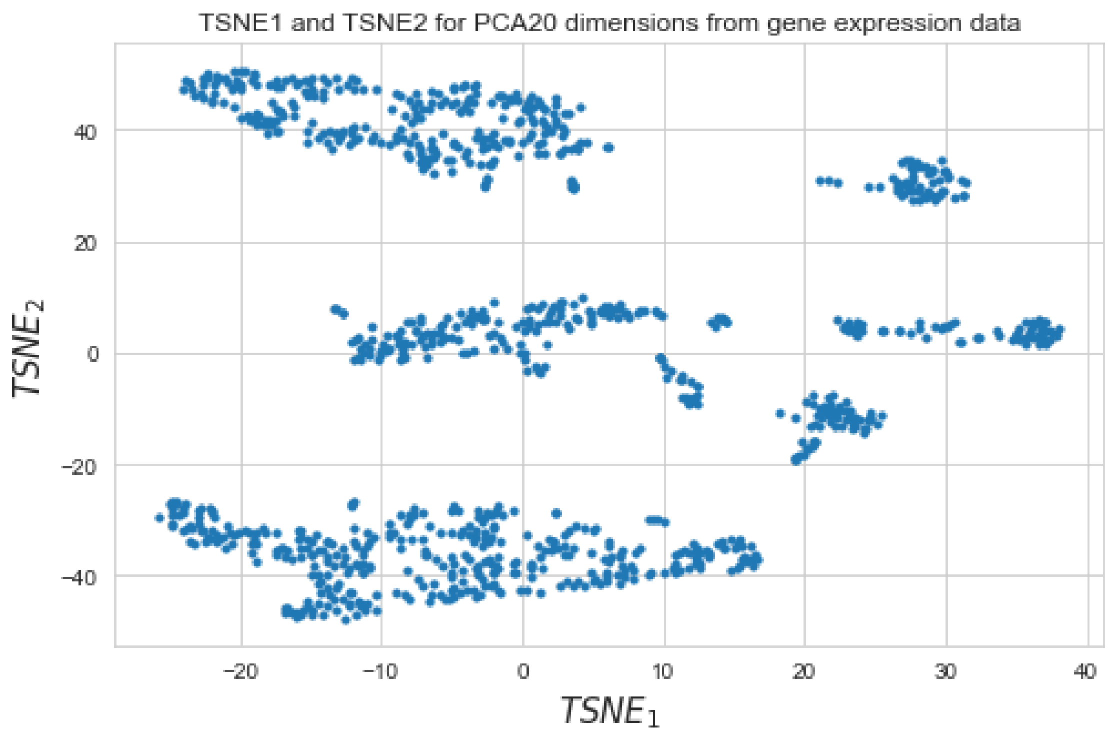
[60]:
```
### Your code here
tsne = TSNE()
tsne_pca_results = tsne.fit_transform(principalDf_20)
```

```
/Users/davitbarblishvili/opt/anaconda3/lib/python3.9/site-
packages/sklearn/manifold/_t_sne.py:780: FutureWarning: The default
initialization in TSNE will change from 'random' to 'pca' in 1.2.
  warnings.warn(
/Users/davitbarblishvili/opt/anaconda3/lib/python3.9/site-
packages/sklearn/manifold/_t_sne.py:790: FutureWarning: The default learning
rate in TSNE will change from 200.0 to 'auto' in 1.2.
  warnings.warn(
```

Q11. Plot the data (first 20 principal components) projected onto the first two t-SNE dimensions.

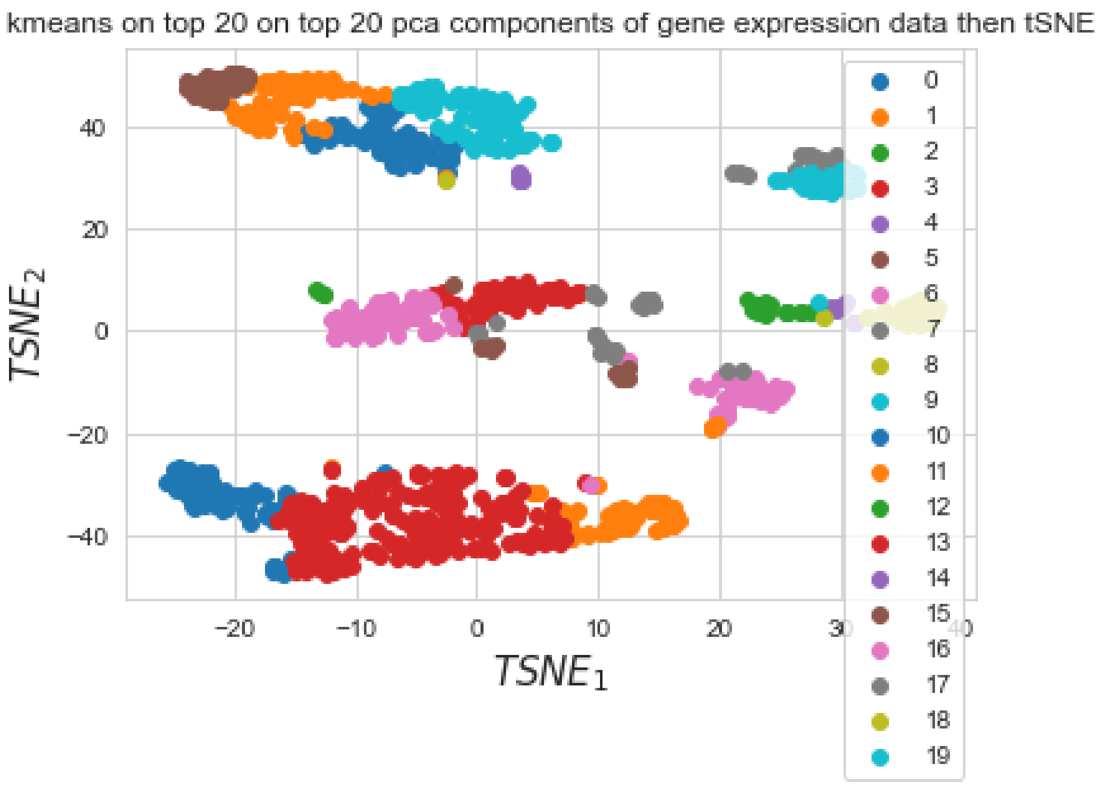
[61]:
```
### Your code here
plt.figure(figsize=(8, 5))
plt.scatter(tsne_pca_results[:, 0], tsne_pca_results[:, 1], s=10)
plt.xlabel('$TSNE_1$', fontsize=15)
plt.ylabel('$TSNE_2$', fontsize=15)
plt.title('TSNE1 and TSNE2 for PCA20 dimensions from gene expression data')
```

[61]:
```
Text(0.5, 1.0, 'TSNE1 and TSNE2 for PCA20 dimensions from gene expression data')
```



Q12. Plot the data (first 20 principal components) projected onto the first two t-SNE dimensions, with points colored by their cluster assignments from part 2.

```
[62]: ### Your code herefor i in np.unique(pca_labels):
      from matplotlib.pyplot import figure

      for i in np.unique(pca_labels):
          plt.scatter(tsne_pca_results[pca_labels == i , 0] ,␣
        ↪tsne_pca_results[pca_labels == i , 1] , label = i)
      plt.xlabel('$TSNE_1$', fontsize=15)
      plt.ylabel('$TSNE_2$', fontsize=15)
      plt.title('kmeans on top 20 on top 20 pca components of gene expression data␣
        ↪then tSNE')
      plt.legend()
      plt.show()
```



Q13. Why is there overlap between points in different clusters in the t-SNE plot above?

5.4.1 Your answer here

There is overlap because we are reducing an already reduced dimensionality of PCA further, so we are unable to see the 'depth' or third/more dimensions which may be separating the dataset. Also, tSNE is a probabilisitic algorithm so it's possible that the overlap is due to the probability based nature - which lends itself to non-clear cut /black and white slices.

These 20 clusters may correspond to various cell subtypes or cell states. They can be further investigated and mapped to known cell types based on their gene expressions (e.g. using the K-means cluster centers). The clusters may also be used in downstream analysis. For instance, we can monitor how the clusters evolve and interact with each other over time in response to a treatment.