project2

November 18, 2024

0.1 Titanic Dataset Unsupervised Methods Exploration

This project attempts to explore and understand unsupervised Machine Learning techniques using a dataset, sourced from kaggle, which collects information about passengers aboard the RMS Titanic ocean liner. A number of machine learning concepts are discussed. This includes the use of matrix factorization techniques for transforming the data and multiple clustering models, as well as methods of analysing clustering performance.

0.1.1 What Are We Trying to Learn?

The goal of this unsupervised methods project will be to develop multiple representations of the dataset and to interpret clustering behavior changes via those transformations through visual analysis, clustering metrics, and ground truth. Transformation methods include preprocessing, Singular Value Decomposition, and Nonnegative Matrix Factorization dimension reductions. Clustering methods will include Heirarchical (agglomerative), K-means, Gaussian Mixture, and Spectral Clustering models. A pipeline is constructed to streamline the process of hyperparameter selection and results analysis.

0.1.2 Why Does It Matter?

This project is for the CU Boulder CSPB machine learning course. While regular course material is both relevent and plentiful, the realm of machine learning techniques and tools is incredibly vast and cannot be covered only by reading and standard homework material. An opportunity to explore is invaluable, because it involves attempting to answer unexpected questions and grows experience in a way which is more personable.

0.1.3 Data Link and Info

The Complete Titanic Dataset was sourced from kaggle, and has been uploaded by Vinicius Barbosa Paiva. Little information is provided on the assiciated kaggle page, although many different iterations of the titanic dataset can be found there. The table, provided via Microsoft Excel document, contains 1309 observations of 14 total features including the 'survived' column used as ground truth for the project. Typical of records from the era the completeness of the features is poor, ranging from a single missing value to only a small proportion being filled, and formatting is also necessary for several of them. The feature listing provides a quick explanation of the different columns and how they are encoded.

Feature Listing

1. survival - Survival status 0) No, 1) Yes

2. pclass - Passenger Class 1) 1st, 2) 2nd, 3) 3rd
3. name - Passenger Name
4. sex - Sex ("male", "female")
5. age - Age (years)
6. sibsp - Number of siblings or spouses aboard
7. parch - Number of parents or children aboard
8. ticket - Ticket number
9. fare - Passenger fare paid (British Pounds)
10. cabin - Assigned cabin or cabins
11. embarked - Port of embarkation 'C') Cherbourg, 'Q') Queenstown, 'S') Southampton
12. boat - Lifeboat used if survived
13. body - Body number assigned if recovered
14. home.dest - Declared passenger destination

0.1.4 Imports, Basic Info, and Some Example Observations

[1]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import copy
import scipy as sp
import scipy.stats as stats
import time
import itertools
from sklearn.model_selection import GridSearchCV
from sklearn.decomposition import NMF, TruncatedSVD
from sklearn.cluster import AgglomerativeClustering, KMeans,␣
 ↪SpectralClustering, OPTICS
from sklearn.mixture import GaussianMixture
from sklearn.metrics import (accuracy_score, confusion_matrix,␣
 ↪homogeneity_score,
                            completeness_score, v_measure_score,␣
 ↪adjusted_rand_score,
                            adjusted_mutual_info_score, silhouette_score)
from scipy.cluster.hierarchy import dendrogram
from sklearn.preprocessing import StandardScaler, MinMaxScaler

df = pd.read_excel("data/titanic3.xls")
RANDOMSTATE = 42
NCLUST = 2
```

[2]:
```
df.info()
df.head()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1309 entries, 0 to 1308
```

```
Data columns (total 14 columns):
 #   Column     Non-Null Count  Dtype  
---  ------     --------------  -----  
 0   pclass     1309 non-null   int64  
 1   survived   1309 non-null   int64  
 2   name       1309 non-null   object 
 3   sex        1309 non-null   object 
 4   age        1046 non-null   float64
 5   sibsp      1309 non-null   int64  
 6   parch      1309 non-null   int64  
 7   ticket     1309 non-null   object 
 8   fare       1308 non-null   float64
 9   cabin      295 non-null    object 
 10  embarked   1307 non-null   object 
 11  boat       486 non-null    object 
 12  body       121 non-null    float64
 13  home.dest  745 non-null    object 
dtypes: float64(3), int64(4), object(7)
memory usage: 143.3+ KB
```

```
[2]:    pclass  survived                                             name     sex  \
    0       1         1                    Allen, Miss. Elisabeth Walton  female   
    1       1         1                   Allison, Master. Hudson Trevor    male   
    2       1         0                     Allison, Miss. Helen Loraine  female   
    3       1         0             Allison, Mr. Hudson Joshua Creighton    male   
    4       1         0  Allison, Mrs. Hudson J C (Bessie Waldo Daniels)  female   

           age  sibsp  parch  ticket      fare    cabin embarked boat   body  \
    0  29.0000      0      0   24160  211.3375       B5        S    2    NaN   
    1   0.9167      1      2  113781  151.5500  C22 C26        S   11    NaN   
    2   2.0000      1      2  113781  151.5500  C22 C26        S  NaN    NaN   
    3  30.0000      1      2  113781  151.5500  C22 C26        S  NaN  135.0   
    4  25.0000      1      2  113781  151.5500  C22 C26        S  NaN    NaN   

                             home.dest  
    0                     St Louis, MO  
    1  Montreal, PQ / Chesterville, ON  
    2  Montreal, PQ / Chesterville, ON  
    3  Montreal, PQ / Chesterville, ON  
    4  Montreal, PQ / Chesterville, ON  
```

0.1.5 Preliminary Considerations

Several features will need to be reencoded under a new scheme in order to see use. Although several columns will not be utilized for data analysis, missing values exist in several which will. These are thankfully fairly rare in those cases. Relative to other examples for the Titanic, this particular dataset appeared to be more complete than others found. However, the observations are not comprehensive. Only information of about 59% of the original 2224 passengers is available, with 70% of the actual 710 survivors included.

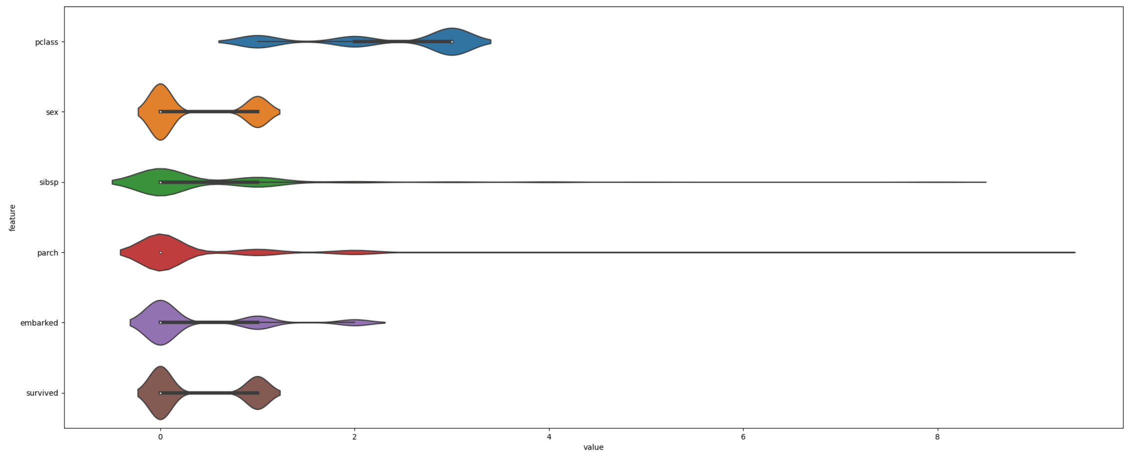
0.1.6 Cleaning and Inclusion Choices

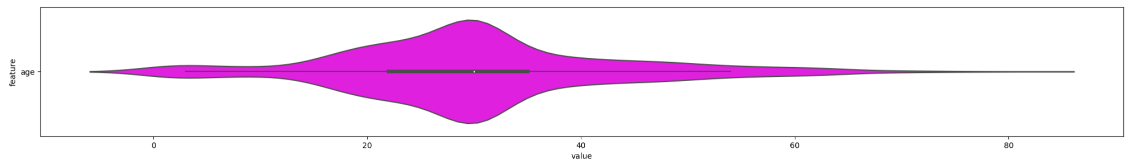
To start, originally nonnumeric features such as 'sex' and 'embarked' have been encoded as integers in order to make use of them. The 'embarked' feature also happened to be missing values in two cases, and so the most heavily occuring option was used to impute with there. The one missing feature in 'fare' is replaced with the median for the feature, which seems reasonable for a monetary value. The most heavily missing feature still included (263 missing), and thereby most in need of imputation, was 'age'. Since there were so many ages to replace, to reduce distortion an average age is calculated for 'sex' value of the observations. This seems reasonable, as the distributions of the two subsets have a different average by two years. Overall, we will see that age is not correlated with the 'survived' feature we will be using for ground truth. Of course, several features are necessary to exclude completely in this case. Passenger 'name' and 'home.dest' will not be helpful. The 'boat' and 'body' columns need to be excluded, because they represent information from after-the-fact with respect to 'survived', only including additional noise as they are very incomplete. This leaves the remaining decision for 'cabin'. Less than one quarter of the column is populated, and it has been left out largely because it seems too far gone. Supporting this though is the fact that although likelihood of survival is heavily associated with passenger class the cabins of a given class are not strictly tied to a given deck (where A-deck would have had best proximity to the lifeboats).

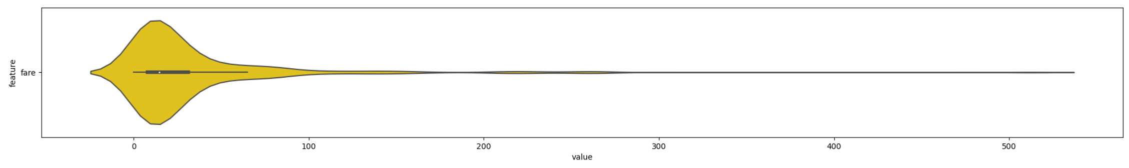
Violin plots are provided prior to a correlation matrix, with some consideration of the correlation matrix discussed below. The distributions shown with violins helps illustrate which features are categorical and which numeric. The significant distinction in range for the 'fare' and 'age' features also justifies why thorough use of scaling is employed for the project in the subsequent section. The 'fare' feature's distribution is the most extreme, and ranges from a single passenger who supposedly paid nothing for their ticket to several passengers who paid over 200 pounds. Also, set of four who supposedly paid over 500 pounds, which at first I suspect to be a transcription error but after some looking found claims that first class could potentially cost 870 pounds.

[3]:
```
td = df[['pclass',
         'sex',
         'age',
         'sibsp',
         'parch',
         'fare',
         'embarked',
         'survived']].copy()

td.sex = td.sex.map({'male': 0, 'female': 1})
td.age = td.age.fillna(td.groupby('sex')['age'].transform('mean'))
td.fare = td.fare.fillna(td.fare.median())
td.embarked = td.embarked.fillna('S').map({'S': 0, 'C': 1, 'Q': 2})
gtruth = df.survived



plt.figure(figsize=(25, 10))
```

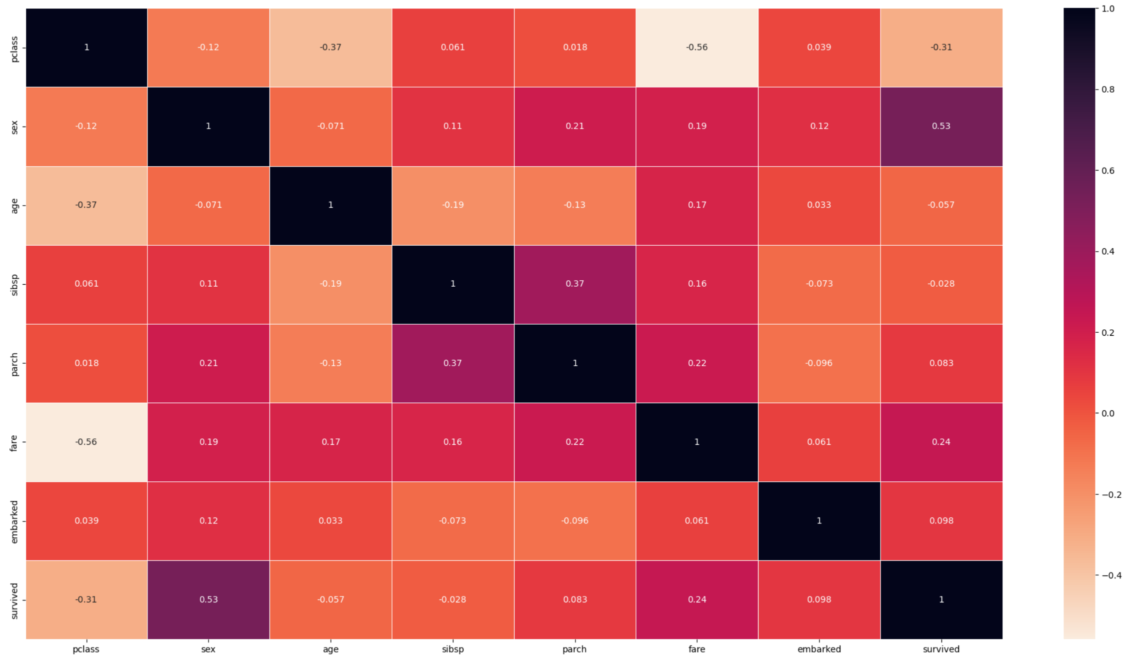
```
sns.violinplot(x="value", y="feature", data=td.drop(['age','fare'],axis=1).
  ↪melt(var_name="feature", value_name="value"))
plt.show()
plt.figure(figsize=(25, 3))
sns.violinplot(x="value", y="feature", data=td[['age']].
  ↪melt(var_name="feature", value_name="value"), color='magenta')
plt.show()
plt.figure(figsize=(25, 3))
sns.violinplot(x="value", y="feature", data=td[['fare']].
  ↪melt(var_name="feature", value_name="value"), color='gold')
plt.show()



plt.figure(figsize=(25, 13))
sns.heatmap(td.corr(), annot=True, cmap='rocket_r', linewidth=.5)

corr_order = list(td.corr().iloc[:,-1].abs().sort_values(ascending=False)[1:].
  ↪index)
td = td[corr_order] # drops the survived column from titanic data
td.head()
```







```
[3]:    sex  pclass      fare  embarked  parch      age  sibsp
     0    1       1  211.3375         0      0  29.0000      0
     1    0       1  151.5500         0      2   0.9167      1
     2    1       1  151.5500         0      2   2.0000      1
     3    0       1  151.5500         0      2  30.0000      1
     4    1       1  151.5500         0      2  25.0000      1
```



We can see from our correlation matrix some interesting associations. The most associated with survival being ‘pclass’ and ‘sex’. What would have been more ideal to the project then might have been at least one other feature available which did not itself associated heavily with ‘pclass’ and ‘sex’, unlike what we see with ‘fare’. This association makes ‘fare’ rather redundant, although we will be attempting to cut through some redundancy in our features via dimension reduction.

0.1.7 Feature Transformations, SVD and NMF

Matrix factorization is a technique in which a matrix is decomposed into a product of two or more smaller matrices, each representing different latent features or factors. This approach is commonly used for “dimensionality reduction” because it transforms a dataset of potentially very many features into a lower dimensional representation, ideally capturing the most informative structure of the original data. By projecting data onto a smaller space, matrix factorization uncovers patterns and relationships that may not be apparent in the higher dimensional space, often making data easier to analyze, visualize, or use in downstream tasks like the clustering we will be doing later in this project. As can be learned from Principal Component Analysis, while the key objective of reducing dimensions is to improve useability, this reduction also comes at the cost of some (often quantifiable) amount of information from the data. The further information loss caused by the reduction, the less a reduced dataset is distinguishable. The practice in this context is then an attempt to balance the trade-off between improved useability and any loss of information. This is interpreted by the ratio of the retained "explained variance" of the reduction, or "reconstruction error" for NMF.

A Little Additional Feature Engineering Some small success has also been seen by improving performance later on in the project by way of engineering features of higher correlation with survival here. By exploring the effects of additional scaling, used in conjunction with the feature scaling typical for dimensional projection techniques, datapoint alignment can be altered in a way which leads to some improvement of outcomes for our clustering algorithms later. Notice in the following code block that further scaling of 'sex' and 'pclass' columns not only keeps feature scales in rough proximity to one another, but can shift relative positioning around and even influence the spread among groupings. Projection of a dataset to lower features by way of matrix factorization can be described as multiple rotations or reflections combined with rescaling. In short, by selecting columns and scaling their values from a range between zero and one down to between zero and one-half we can opt for an overall rotation which aligns datapoint positions which is at least somewhat more aggreable to our clustering algorithms. For further illustration of the effects, see the tangential notebook included with the submission. The effects are interesting, but the illustration was overzealous to include in the primary presentation.

Singular Value Decomposition (SVD) Described in Mining of Massive Datasets by Leskovec, Rajaraman, Ullman (Section 11.3), SVD is a matrix factorization technique that decomposes a matrix $ A $ into three matrices: $ A = U V^T $, where: 1. $ U $ is an $ m ×r $ column-orthonormal matrix ; that is, each of its columns is a unit vector and the dot product of any two columns is $ 0 $. 2. $ V $ is an $ n ×r $ column-orthonormal matrix. Note that we always use $ V $ in its transposed form, so it is the rows of $ V^T $ that are orthonormal. 3. $ $ is a diagonal matrix; that is, all elements not on the main diagonal are $ 0 $. The elements of $ $ are called the singular values of $ A $.

The largest singular values and their corresponding vectors (returned as the columns of $ V $ in TruncatedSVD) capture the most variance in the data. By keeping only the top $ k $ singular values, we obtain a low-rank approximation of the matrix that captures the most essential information, effectively reducing dimensionality.

To apply the interpretation discussed by Leskovec, Rajaraman, Ullman; the values of a column of $ V $ represent the "concept" (loading) which aligns closely with the features we use to represent the observations held in our dataset, in this case representing Titanic passengers. The weighting provided by this concept can help us to transform our existing observations down to an individual value per observation. We can then progress to a total number of dimensions (transformed features) strictly less than the original number in the dataset in the case of TruncatedSVD.

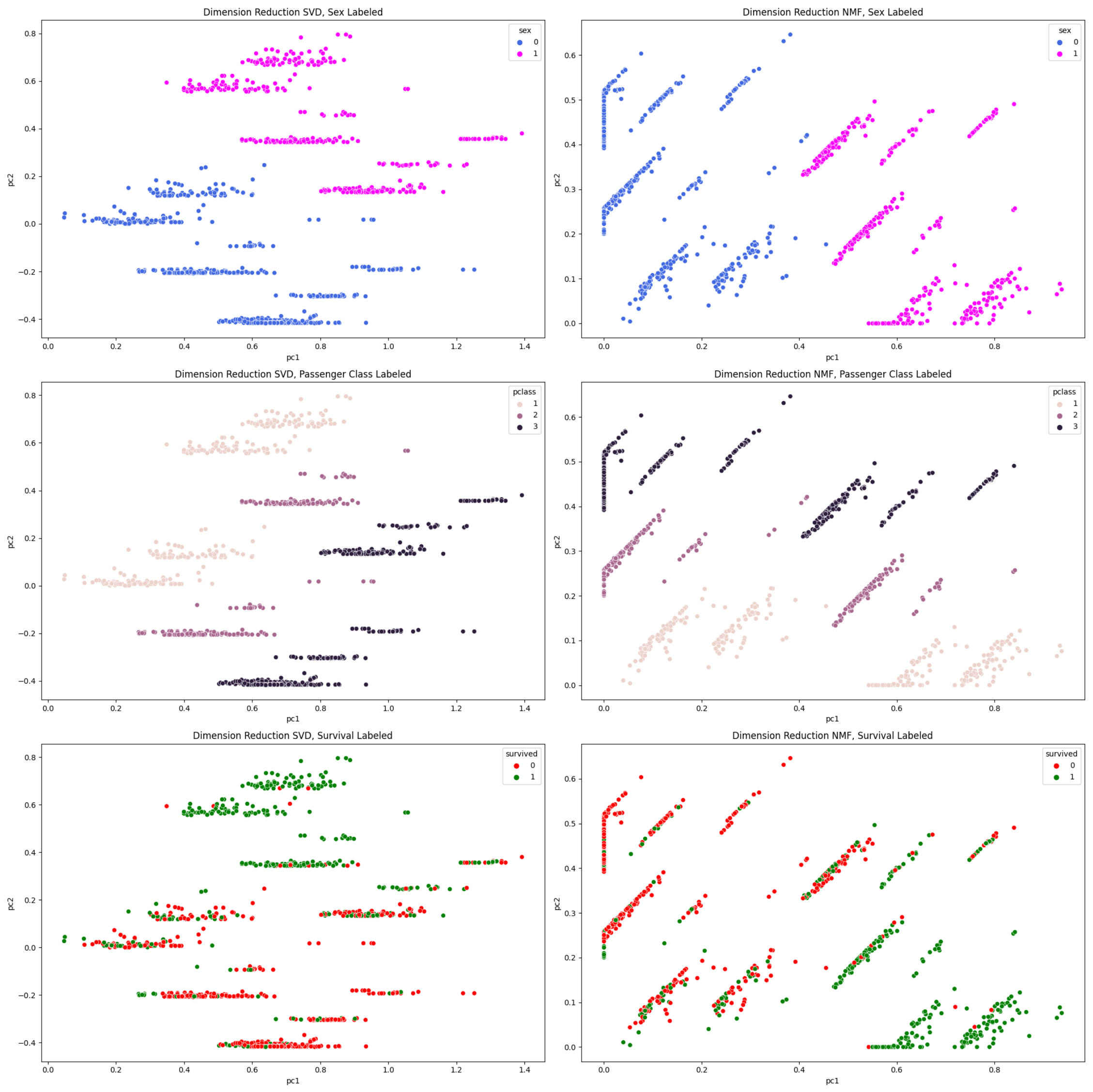
Because each concept is a balance between features in the orignial dataset, it is common to pre-process values prior to dimensionality reduction. Where the magnitude or variance of values for a particular feature are significantly greater than others, they can result in weights which dominate the linear combination we want to produce with the features. To control for this, using SVD typically involves centering the dataset so that each feature has zero for its mean and that its variance is scaled to a standard deviation of one. This would mean, of course, that the result of StandardScalar and SVD will tend have both positive and negative values.

Non-negative Matrix Factorization (NMF) You may have noticed that our original dataset contains no negative values. Its features are primarily examples of categorical values, counts, ordinal values, names, prices, etc. By existing within this constraint, we have the opportunity to an alternative route for dimensionality reduction in our data. NMF decomposes a non-negative matrix $ A $ into two smaller non-negative matrices, $ W $ and $ H $, such that $ A WH $. The decomposition of $ A $ optimizes the distance $ d $ between $ A $ and product $ WH $, typically with the distance function Frobenius norm, an extension of Euclidean norm. Once the objective function has been minimized for NMF the matrix $ W $ represents the transformed data and $ H $ represents the components (loadings). Where we wish to perform further work, say clustering on a reduced version of our original dataset, we take $ k $ columns of $ W $ representing the lower dimensional projection. With sklearn, the number of components is specified in advance, which serves our needs when examining our ground truth.

To correspond with the nonnegativity requirement of NMF it is common practice to scale features to a range of zero to one, which helps prevent a feature of large magnitude or variance from dominating the resulting components. Also of note, is that NMF algorithms use an iterative method (typically using coordinate descent) to alternate between optimizing the values of $ W $ or $ H $ while the other is held fixed. This only qualifies as an approximation of the NMF problem itself, and where because producing a full solution rather than an approximation with SVD contributes to the interpretation that NMF is inferior to SVD.

[4]:
```
X_std = StandardScaler(with_mean=True, with_std=True).fit_transform(td)
X_zo = MinMaxScaler(feature_range=(0, 1)).fit_transform(td)

# selecting for rotations in the projecions
X_zo[:, 1] = np.vectorize(lambda x: x/1.4)(X_zo[:, 1]) # 'sex'
X_zo[:, 0] = np.vectorize(lambda x: x/1.4)(X_zo[:, 0]) # 'pclass'

X_svd = pd.DataFrame(TruncatedSVD(n_components=2, algorithm='arpack',␣
 ↪random_state=RANDOMSTATE).fit_transform(X_zo),
                     columns=['pc1', 'pc2'])
X_nmf = pd.DataFrame(NMF(n_components=2, init='random',
                         beta_loss='frobenius', solver='cd',
                         random_state=RANDOMSTATE).fit_transform(X_zo),
                     columns=['pc1', 'pc2'])

fig, axes = plt.subplots(3, 2, figsize=(20, 20), sharey=False)

sns.scatterplot(x='pc1', y='pc2', data=X_svd, hue=td.sex, palette={0:␣
 ↪'royalblue', 1: 'magenta'}, ax=axes[0][0])
axes[0][0].set_title('Dimension Reduction SVD, Sex Labeled')
```

```
sns.scatterplot(x='pc1', y='pc2', data=X_nmf, hue=td.sex, palette={0:␣
 ↪'royalblue', 1: 'magenta'}, ax=axes[0][1])
axes[0][1].set_title('Dimension Reduction NMF, Sex Labeled')

sns.scatterplot(x='pc1', y='pc2', data=X_svd, hue=td.pclass, ax=axes[1][0])
axes[1][0].set_title('Dimension Reduction SVD, Passenger Class Labeled')

sns.scatterplot(x='pc1', y='pc2', data=X_nmf, hue=td.pclass, ax=axes[1][1])
axes[1][1].set_title('Dimension Reduction NMF, Passenger Class Labeled')

sns.scatterplot(x='pc1', y='pc2', data=X_svd, hue=gtruth, palette={0: 'red', 1:␣
 ↪'green'}, ax=axes[2][0])
axes[2][0].set_title('Dimension Reduction SVD, Survival Labeled')

sns.scatterplot(x='pc1', y='pc2', data=X_nmf, hue=gtruth, palette={0: 'red', 1:␣
 ↪'green'}, ax=axes[2][1])
axes[2][1].set_title('Dimension Reduction NMF, Survival Labeled')

plt.tight_layout()
plt.show()
```



0.1.8 Projected Dataset Visualization

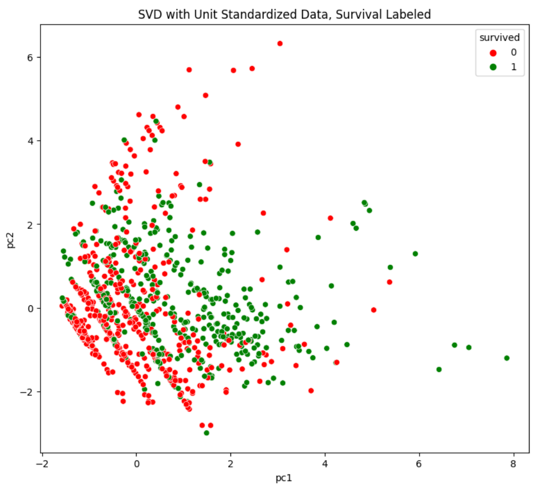
Reducing dimensions has afforded us the ability to see the results plotted above. This is important as it helps us to explore the results more directly, and can serve as a tool later where we want a visual for how different clustering algorithms differed in their classifications. Taking some time to interpet these plots is helpful for a number of the upcoming decisions in the project. Where we colorize datapoints to explore where and how the projection is separable, we see that the use of zero-one scaling for our dataset works fairly well. Also note the particular quirk of NMF where any negative value resulting from the factorization process is automatically replaced by zero, causing some datapoints to collapse against the proverbial boundary. Although, from the bottom row of scatter plots where survival is colored in we can see how intermixed the two classes are. We will have to see how well clustering models will be able to latch onto the datapoints, and just how much performance might be altered where we try to use projected data.

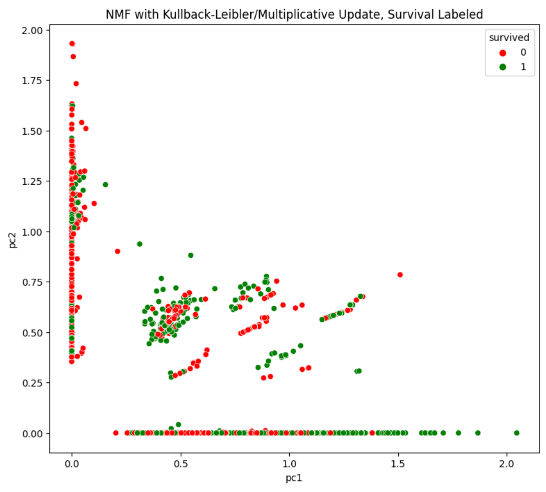
Both SVD and NMF plots are showing projections of the zero-one scaled data, produced using MinMaxScaler - denoted 'X_zo'. Although it is more common to use data with mean-zero and standard deviation scaled to one with SVD, a data transformation subset made using StandardScaler here named 'X_std', exploring different options showed better results among clustering algorithms for using 'X_zo' with either data projection.

Similarly, a variety of choices for solver and beta_loss have been considered as well as methods for preprocessing. The following illustrates the outcomes that can be found from some of these combinations, but defaults such as 'cd' (coordinate descent) and 'frobenius' have won out with the preprocessed choices at later stage of examining clustering algorithm performance. The ideal with clustering is generally separable, homogenous clusters. It should be evident from the visualizations here that we can actually do worse.

```
[5]: svd_w_std = pd.DataFrame(TruncatedSVD(n_components=2, algorithm='arpack',␣
      ↪random_state=RANDOMSTATE).fit_transform(X_std),
                            columns=['pc1', 'pc2'])
     nmf_w_kl = pd.DataFrame(NMF(n_components=2, init='random',
                                 beta_loss='kullback-leibler', solver='mu', # using␣
      ↪multiplicative update approximation
                                 random_state=RANDOMSTATE).fit_transform(X_zo),
                            columns=['pc1', 'pc2'])

     fig, axes = plt.subplots(1, 2, figsize=(20, 8), sharey=False)

     sns.scatterplot(x='pc1', y='pc2', data=svd_w_std, hue=gtruth, palette={0:␣
      ↪'red', 1: 'green'}, ax=axes[0])
     axes[0].set_title('SVD with Unit Standardized Data, Survival Labeled')

     sns.scatterplot(x='pc1', y='pc2', data=nmf_w_kl, hue=gtruth, palette={0: 'red',␣
      ↪1: 'green'}, ax=axes[1])
     axes[1].set_title('NMF with Kullback-Leibler/Multiplicative Update, Survival␣
      ↪Labeled')
```

```
[5]: Text(0.5, 1.0, 'NMF with Kullback-Leibler/Multiplicative Update, Survival
     Labeled')
```



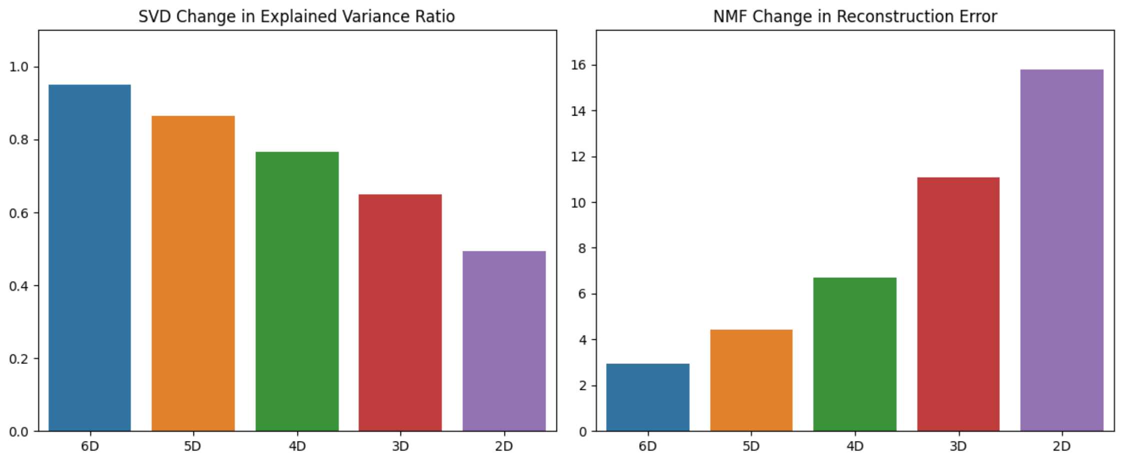


0.1.9 Interpreting Matrix Factorization Performance

As a brief aside with matrix factorization, we should consider their existing metrics for interpreting the quality of dimension reduction. However, SVD and NMF do not share the same metric for this purpose, and cannot be compared directly. However, taking the opportunity to consider them is still important once we have moved forward with investigating the performance results of our clustering algorithms. We can potentially (depending on any prior exposure) generate some expectation for what it may look like to use different transformations of the data with those algorithms because of these metrics. Also, these plots were helpful as a sort if intermediary check between the dataset transformation section above and the clustering performance gauntlet (elbow plot) used below. If a change to SVD or NMF data produced noticeble negative changes here, it proved to also be negative down there.

In both cases, we use a conservative projection from our seven original features down to six. This seems a reasonable starting point to ensure that the loss of information associated with reduction is kept small. We then compare this result with that of reducing to fewer dimensions and ending in two, the amount we want to in order to be able to visually plot our data (though as we will see is not always entirely necessary). Considering the left bar graph, we can see that SVD used on the unit standardized data drops in its cumulative explained variance to about 49% for only two dimensions. This is not a particularly good sign for using the SVD transformed data, which would ideally have reduced much less. The NMF comparison on the right behaves in opposite fashion. Reconstruction error, used by NMF, is the sum of squared differences between the original matrix and its approximation. This is the Frobenius norm, which is similar to how Residual Sum of Squares (RSS) is used for measuring error in regression. Additional loss of information latent to components is conveyed then by an increase in the value of the reconstruction error. For our NMF reduction to two dimensions we see the error increase by about a factor of five. How bad is this amount of change? At least relative to the upcoming goal of clustering our data, we will see that projection with NMF provides a slightly worse performance hit. This inclines a conclusion that NMF results were a bit worse than SVD in this case.

[6]:
```
svd_exp = []
nmf_rec = []

for i in range(6,1,-1):
    svd_exp.append(sum(TruncatedSVD(n_components=i, algorithm='arpack',␣
 ↪random_state=RANDOMSTATE)
                    .fit(X_std)
                    .explained_variance_ratio_))
    nmf_rec.append(NMF(n_components=i, max_iter=400, init='random',␣
 ↪random_state=RANDOMSTATE).fit(X_zo)
                    .reconstruction_err_)

x = ["6D", "5D", "4D", "3D", "2D"]

fig, axes = plt.subplots(1, 2, figsize=(12, 5), sharey=False)

sns.barplot(x=x, y=svd_exp, ax=axes[0])
axes[0].set_title('SVD Change in Explained Variance Ratio')
axes[0].set_ylim(0, 1.1)

sns.barplot(x=x, y=nmf_rec, ax=axes[1])
axes[1].set_title('NMF Change in Reconstruction Error')
axes[1].set_ylim(0, 17.5)

plt.tight_layout()
plt.show()
```



4. Model building / Model choice (15 pts)
   How many models have been used and compared? Any reasonings for the model choice?

[7]:
```
param_grids = {
    'AgglomerativeClustering': {'n_clusters': [NCLUST],
                                'metric': ['euclidean', 'manhattan', 'cosine'],
                                'linkage': ['complete', 'average', 'single']},

    'KMeans': {'n_clusters': [NCLUST],
               'init': ['k-means++','random'],
               'algorithm': ['lloyd','elkan'],
               'n_init': ['auto', 1, 2, 3, 4, 5],
               'random_state': [RANDOMSTATE]},

    'GaussianMixture': {'n_components': [NCLUST],
                        'covariance_type':['full', 'tied', 'diag', 'spherical'],
                        'init_params': ['kmeans', 'k-means++', 'random',␣
 ↪'random_from_data'],
                        'n_init': [1,2,3,4,5],
                        'random_state': [RANDOMSTATE]},

    'SpectralClustering': {'n_clusters': [NCLUST],
                           'affinity': ['rbf'], # 'nearest_neighbors'],
                           'n_init': [3],
                           'gamma': [.01,.1,1,2,4,8],
                           #'n_neighbors': [1,3,30,40,50],
                           'assign_labels': ['kmeans', 'cluster_qr'],
                           'n_jobs':[-1],
                           'random_state': [RANDOMSTATE]}

}


def accuracy_scorer(estimator, X):

    labels = estimator.fit_predict(X)

    cm = confusion_matrix(gtruth, labels)

    perm = cm.argmax(axis=0)

    remap = np.array([perm[i] for i in labels])

    cm1 = confusion_matrix(gtruth, remap)

    return np.trace(cm1) / np.sum(cm1)


def adj_rand_scorer(estimator, X):
    predictions = estimator.fit_predict(X)

    return adjusted_rand_score(gtruth, predictions)


def v_m_scorer(estimator, X):

    predictions = estimator.fit_predict(X)

    return v_measure_score(gtruth, predictions)


def gridsearch(estimator, param_grid, scoring, X):
    '''make the params and run them'''
    keys = param_grid.keys()
    values = param_grid.values()
    combinations = [dict(zip(keys, combo)) for combo in itertools.
 ↪product(*values)]
    scores = []
    for params in combinations:
        scores.append(scoring(estimator(**params), X))

    return {'params': np.array(combinations), 'scores': np.array(scores)}


def pipeline(clst, p_grid, scorer, X, name_X):
    '''pipeline for a given format of cluster alg'''

    search = gridsearch(estimator=clst, param_grid=p_grid, scoring=scorer, X=X)

    idx = search['scores'].argmax()
    best = {key: value[idx] for key, value in search.items()}

    start_time = time.time()
    clst(**best['params']).fit(X)
    end_time = time.time()
    best['time'] = end_time - start_time

    best['model'] = clst.__name__
    best['data'] = name_X

    #print("Highest scored parameter combination:\n", best['params'])
    #print(f"{best['model']}, {scorer.__name__}: ", best['scores'])

    return best

def make_plots(clst, params, X, name_X):
    '''map result to scatter plot for some pipeline outcome'''
    spec = clst(**params).fit_predict(X)

    cm = confusion_matrix(gtruth, spec)
    perm = cm.argmax(axis=0)

    remap = np.array([perm[i] for i in spec])

    fig, axes = plt.subplots(1, 2, figsize=(20, 7), sharey=False)

    sns.scatterplot(x='pc1', y='pc2', data=X_svd, hue=remap, palette={0: 'red',␣
↪1: 'green'}, ax=axes[0])
    axes[0].set_title(f'{clst.__name__}, {name_X}, Clusters Labeled')

    sns.scatterplot(x='pc1', y='pc2', data=X_svd, hue=gtruth, palette={0:␣
↪'red', 1: 'green'}, ax=axes[1])
    axes[1].set_title(f'SVD projection Reference Scatterplot, Ground Truth␣
↪Labeled')
```

0.1.10 Hyperparameter Searching and Performance Analysis

From here the project transitions to explore clustering algorithms. However the effort so far is not left out moving forward. There is still some light left to be shed on what we saw with dimension reduction. Where we use preprocessing and/or matrix factorization with data beforehand, how does this impact another unsupervised method such as clustering? We have already seen from visualizations that our ground truth was not able to be fully separable (it is still quite intermixed to be honest) after any of our transformations. Can any clustering method we might choose manage to learn these classes, even somewhat?

We need to spend time exploring the available hyperparameters of our clustering methods, but we can also incorporate the differently transformed versions of our data in order to potentially make further conclusions about them as well as dimensionality reduction. To start, we need to use a metric which will prioritize some form of clustering performance. We also need a way to process through hyperparameters which works with clustering our whole dataset. The whole of the dataset is used because clustering methods work fairly differently as unsupervised methods (they have no 'predict' method), and under the expectation that we would not need to evaluate a model in terms of categorizing some future Titanic passengers.

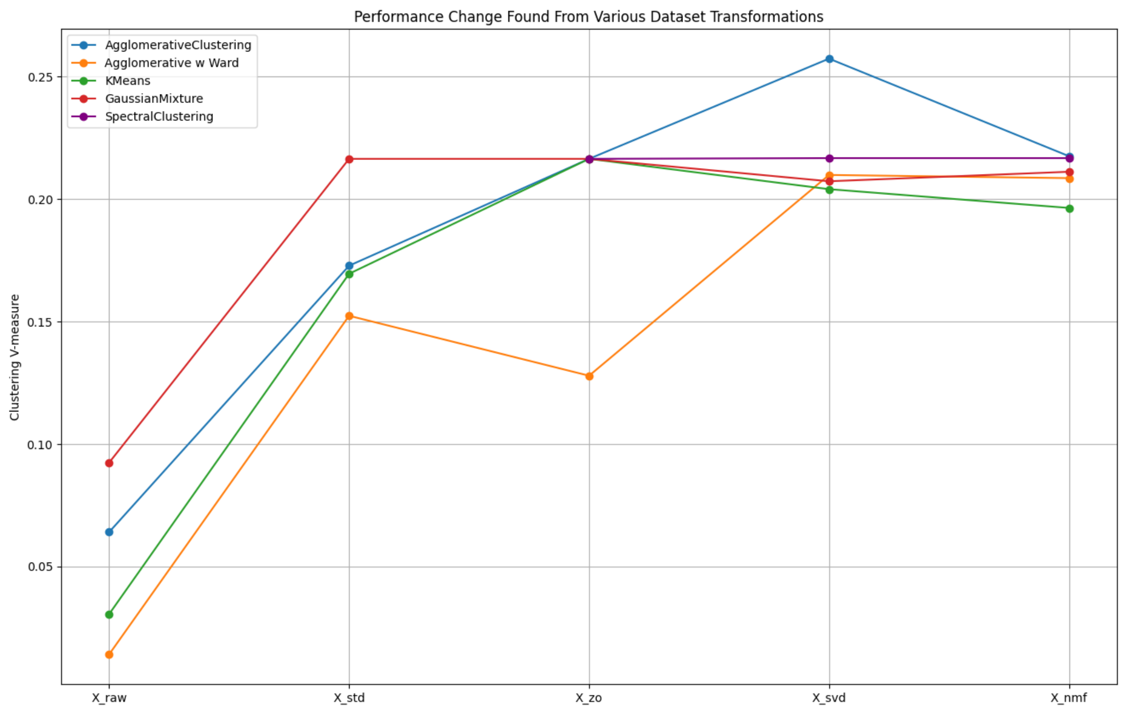
To accomplish this the project uses a custom attempt at parameter grid search, called 'gridsearch'. It uses a prespecified estimator to sequence through the various combinations available of the hyperparameters provided in a dictionary of possible hyperparameter values. This search function compiles a listing of which parameter combinations were used and their scores from the estimator. The pipeline function, 'pipeline', calls the search and determines its top performer, before evaluating its associated fit time, and returns a dictionary including the parameters, score, which clustering algorithm, fit time, and which dataset was used to achieve this performance. The four clustering algorithms the project explores are searched across five representations of the dataset, with outcomes plotted in the line chart below.

[8]:
```
all_my_exes = {'X_raw': td,'X_std': X_std, 'X_zo': X_zo, 'X_svd': X_svd,␣
 ↪'X_nmf': X_nmf}

agglom_list = []
kmeans_list = []
gauss_list = []
agglom_ward = []

for name, X in all_my_exes.items():
    agglom_list.append(pipeline(AgglomerativeClustering,
                                param_grids['AgglomerativeClustering'],
                                v_m_scorer,
                                X,
                                name))
    kmeans_list.append(pipeline(KMeans,
                                param_grids['KMeans'],
                                v_m_scorer,
                                X,
                                name))
    gauss_list.append(pipeline(GaussianMixture,
                               param_grids['GaussianMixture'],
                               v_m_scorer,
                               X,
                               name))
    agglom_ward.append(pipeline(AgglomerativeClustering,
                   {'n_clusters': [NCLUST], 'metric': ['euclidean'], 'linkage':␣
 ↪['ward']},
                   v_m_scorer, X, name))
```

[9]:
```
spectral_list = []
for name in list(all_my_exes.keys())[2:]:
    s = pipeline(SpectralClustering, param_grids['SpectralClustering'],␣
 ↪v_m_scorer, all_my_exes[name], name)
    spectral_list.append(s)
    #print(s)
```

[10]:
```
clst_names = ["AgglomerativeClustering", "Agglomerative w Ward", "KMeans",␣
 ↪"GaussianMixture"]
cats = list(all_my_exes.keys())
scores = pd.DataFrame([[agglom_list[i]['scores'] for i in range(len(all_my_exes.
 ↪keys()))],
                       [agglom_ward[i]['scores'] for i in range(len(all_my_exes.
 ↪keys()))],
                       [kmeans_list[i]['scores'] for i in range(len(all_my_exes.
 ↪keys()))],
                          [gauss_list[i]['scores'] for i in range(len(all_my_exes.
↪keys()))]],
                         columns=cats).T

plt.figure(figsize=(16, 10))
for i in range(len(clst_names)):
    plt.plot(cats, scores[i], label=clst_names[i], marker='o')

plt.plot(cats[2:], [spectral_list[i]['scores'] for i in range(3)],␣
↪label="SpectralClustering", marker='o', color='purple')

plt.grid(True)
plt.ylabel('Clustering V-measure')
plt.legend()
plt.title('Performance Change Found From Various Dataset Transformations')
```

[10]:
```
Text(0.5, 1.0, 'Performance Change Found From Various Dataset Transformations')
```



0.1.11 Search and Dataset Comparisons

The clustering methods the project looks at will be explored in more depth shortly. These are agglomerative clustering, k-means, gaussian mixture, and spectral clustering. As a way to explore the consequences of dimensionality reduction and finalize our feature selection/transformation work we can search for which version bore the most performance. From here, we can use that clustering, dataset pairing when we examine each method and make comparisons. We start with our 'raw' numeric but unscaled data (left). Perhaps unsurprisingly, none of our clustering algorithms can work well with this version. However, performance sees fairly significant uplift in all cases where we scale our data in a way where variance or magnitude are brought into alignment. Generally, algorithms favor zero-one scaled data, but importantly the dropoff in performance from reducing dimensions has turned out to be fairly tame given the results of explained variance and reconstruction error seen earlier. Agglomerative clustering with complete linkage scheme has won out, but using the two-dimensional data produced through SVD. This is considered more later, once we can evaluate more clustering metrics in the comparison stage. To move forward, before making any direct comparisons between the clustering methods themselves or selecting a best choice, we will take some time to examine the algorithms individually in order to explore them more completely.

To evaluate clustering performance the V-measure score is used. It evaluates cluster labeling given some ground truth, and works as an average (the harmonic mean) between the calculated purity (homogeneity) of the clusters and the amount by which members of a given class are assigned together (completeness). In cases where ground truth is not known it can be used as a representation of agreement between two sets of assignments. It attempts to strike a balance in order to better benefit quirks of clustering which can have many unevenly sized assignments, and differs from a metric like accuracy which is predominately influenced by the number of true postives and true negatives. V-measure is best avoided with dataset with too few observations and/or many overall clusters, as random labeling by the algorithm will not bottom out to zero, but we see neither of these issues in our case.

From the chart, five options have been used. this is because the 'ward' linkage scheme is exclusive to the 'euclidean' distance metric with agglomerative clustering and requires its own testing to prevent errors with grid searching. Also, spetral clustering proved uncooperative (convergence problems) when using unscaled numeric data or the full dataset with standardization to unit variance. This likely had to do with some of the parameter combinations searched over, but a compromise of combinations which did not cause problems could not be found and so performance for those dataset versions is excluded.

0.1.12 K-Means

K-means is a partition-based clustering algorithm that iterates between calculating an average (or centroid) of an explicitly set number of clusters for the dataset, and the recalculation of the pairwise distances between datapoints and this iteration's centroids. It works well for cases with clearly defined, spherical clusters. Although, unlike hierarchical clustering, k-means operates on a particular assumption for the number of clusters present.

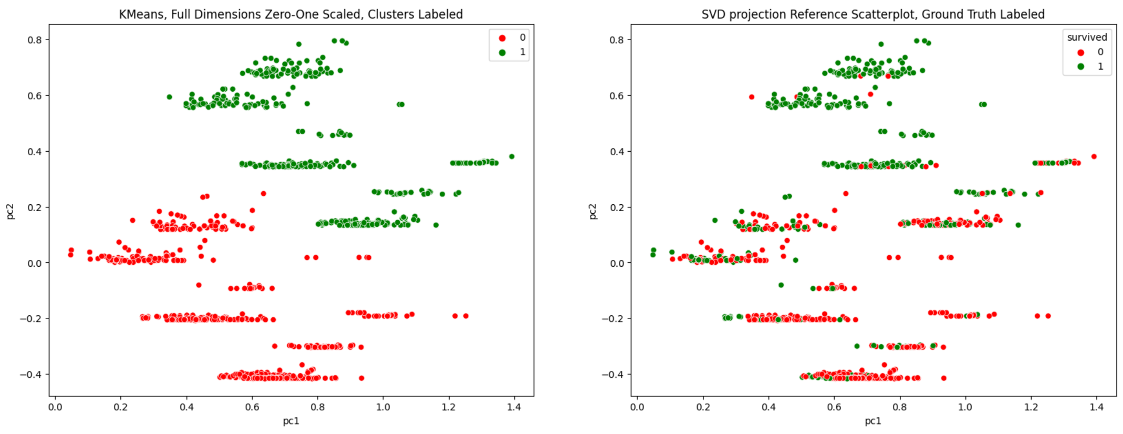
A scheme for initial choices of centroids must be selected for the algorithm, and is important to reduce the likelihood of the algorithm getting "stuck" in a local minimum. For example, the k-means++ initialization scheme selects the initial centroids for the algorithm to improve convergence and achieve a better loss function minimum. Instead of choosing initial centroids randomly, k-means++ starts by selecting the first centroid randomly from the data points. Then, each subsequent centroid is chosen with a probability proportional to the squared distance from the nearest existing centroid. This approach ensures that centroids are spread out, which generally leads to faster convergence and better overall clustering quality.

- **Some Particulars:**
  - Hyperparameters:

    * `n_clusters`: Number of clusters to form.
    * `init`: Method to initialize centroids ("k-means++", or "random").
    * `max_iter`: Maximum number of iterations for convergence.
    * `n_init`: Number of times the algorithm is run with different centroid seeds, returning best outcome.
  - **Pro**: Efficient for large datasets with convex clusters.
  - **Con**: Struggles with non-spherical clusters and sensitivity to initialization.

[11]:
```
kidx = np.array([kmeans_list[i]['scores'] for i in range(len(kmeans_list))]).
  ↪argmax()
kdata = all_my_exes[kmeans_list[kidx]['data']]
make_plots(KMeans, kmeans_list[kidx]['params'], kdata, 'Full Dimensions␣
  ↪Zero-One Scaled')
kmeans_list[kidx]
```

[11]:
```
{'params': {'n_clusters': 2,
  'init': 'k-means++',
  'algorithm': 'lloyd',
  'n_init': 4,
  'random_state': 42},
 'scores': 0.21646534987980354,
 'time': 0.004911899566650391,
 'model': 'KMeans',
 'data': 'X_zo'}
```



K-means makes for the simplest of the algorithms we look at in the project, and demonstrates itself as the fastest. It serves as a good baseline choice for clustering in general, and is even used as a foundation of sorts to be used with some other, more complicated clustering schemes - as we will see shortly. To illustrate clustering assignment outcomes a function 'make_plots' presents the assignments of the algorithm as a side-by-side with the ground truth. Importantly, k-means performed better when we used zero-one scaled data without projection, and those assignments are shown in the left scatter plot.

Our v-measure score for k-means is not low enough to qualify as "random" for it's cluster assignment, but is certainly unsatisfactory. Where we consider zero-one scaling plus projection with SVD (earlier charts) and compare this with the assignments of k-means, it seems clear that the algorithm identified the boundaries separating 'sex' and was unable to optimize in a way that reached closer to the ground truth. There is some predictive justification in this outcome. As we saw early on in the project, 'sex' was fairly highly correlated with 'survival'. K-means generally expects clusters to be globular, so another continuous feature which showed good association might have helped separability in a way that helped the data be more separable to k-means, but for transformations that we have found the small number of categorical options for the most correlated features seems to have made the data too tightly packed and choppy.

0.1.13 Agglomerative Clustering

Agglomerative Clustering is a hierarchical clustering method that starts by treating each data point as its own cluster and then iteratively merges the closest clusters until a stopping criterion is met (often a set number of clusters). This bottom-up approach is suitable for creating nested clusters, allowing a hierarchy of groups based on distances.

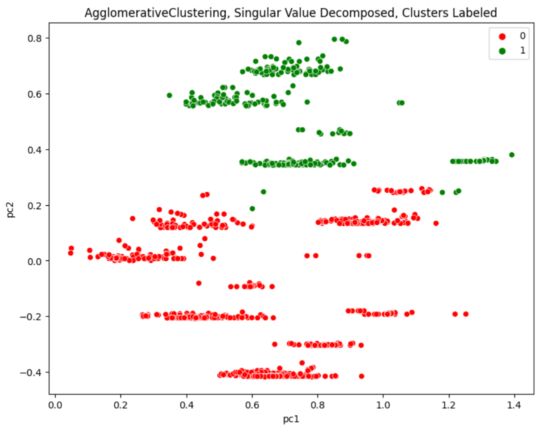
The algorithm uses a choice of various linkage criterions, through which clusters of differing grouping and size are achieved. How distance between points clustered so far is calculated can be performed as an optimization (a min or max) of nearest points between clusters, furthest points between clusters, distance between centers, and even the linkage which results in the lowest variance for the cluster produced by the link in the case of the 'ward' method (only available with euclidean distance metric).

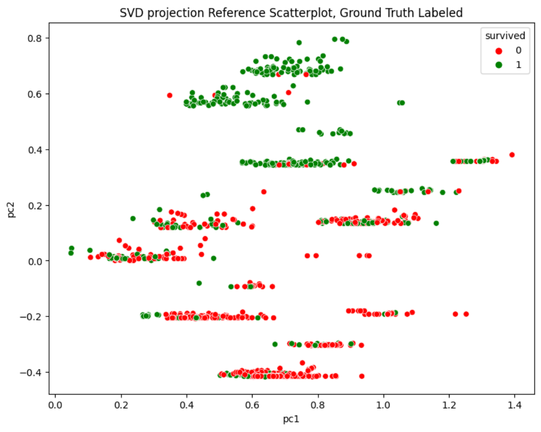
Some Particulars:

- Hyperparameters:
  - `n_clusters`: Number of clusters to form.
  - `metric`: Metric used to compute the linkage (e.g., "euclidean", "manhattan", "cosine").
  - `linkage`: Method to determine distance between clusters ("ward", "complete", "average", "single").
- **Pro**: The variety of linkages provides flexibility for many cluster types.
- **Con**: Computationally expensive for large datasets, as it requires computing all pairwise distances.

[12]:
```
aidx = np.array([agglom_list[i]['scores'] for i in range(len(agglom_ward))]).
 ↪argmax()
adata = all_my_exes[agglom_list[aidx]['data']]
make_plots(AgglomerativeClustering, agglom_list[aidx]['params'], adata,␣
 ↪'Singular Value Decomposed')
agglom_list[aidx]
```

[12]:
```
{'params': {'n_clusters': 2, 'metric': 'euclidean', 'linkage': 'complete'},
 'scores': 0.2573957428215368,
 'time': 0.02121257781982422,
 'model': 'AgglomerativeClustering',
 'data': 'X_svd'}
```





Although not as fast as k-means, for the volume of data being used agglomerative clustering remains quick, yet proves more flexible. Really, agglomerative is a family of methods unto itself where some combination of metric/linkage choice may prove capable of outperforming other cluster methods. We saw in this case that even though datapoints tended to split as several groupings, that better performance was achieved by measuring for largest distance between existing groupings. In fact, agglomerative has been able to at least marginally identify the classes in a way that does not simply stick to separation from 'sex' values.

0.1.14 Gaussian Mixture Model (GMM)

The Gaussian mixture model is a probabilistic model that assumes the data is generated from a mixture of several Gaussian distributions. A Gaussian distribution (or normal distribution) is a continuous probability distribution characterized by its bell-shaped curve, defined by two parameters: the mean (center) and the variance (spread or width). This distribution describes data that clusters around a central value, with fewer instances the farther you go from the mean, following a symmetric pattern.

In a Gaussian mixture model, a "component" represents one Gaussian distribution within the overall mixture, each with its own mean and variance. Each component then describes a subset of data that tends to cluster around that Gaussian's mean. Thus, the concept of a component in a mixture aligns with the idea of clusters in clustering - capturing points that are more similar to one another in terms of their probability density. Multiple components allow the model to better represent anisotropic clusters (they present patterns or similarity in shaping or direction) or clusters of varying density, as each Gaussian can cover different parts of the data, possibly with some overlap between them. This distinguishes GaussianMixture significantly from the previous algorithms. GaussianMixture from sklearn can potentially be used as an extension of k-means, used in order to incorporate information about the covariance structure of the data as well as the centers of the latent Gaussians. By default, it initializes weights and means via k-means itself. Consequently, we must specify a number of components preemptively.

Also important is the option of how covariance is handled for the components, coresponding with the "covariance_type" hyperparameter. Covariance of components corresponds with the characteristics of the clusters, like shape and orientation.

“full”: each component has its own general covariance matrix.

“tied”: all components share the same general covariance matrix.

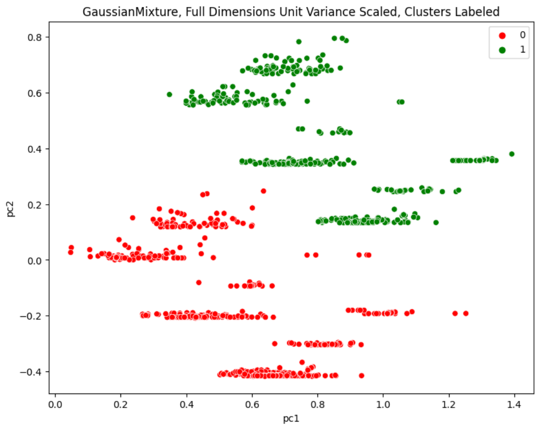
“diag”: each component has its own diagonal covariance matrix.

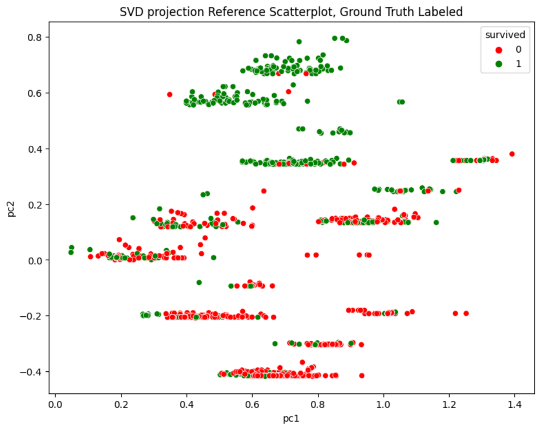
“spherical”: each component has its own single variance.

- **Some Particulars:**
  - Hyperparameters:
    - `n_components`: Number of Gaussian distributions (clusters).
    - `covariance_type`: Shape of covariance (“full”, “tied”, “diag”, “spherical”).
    - `max_iter`: Maximum number of iterations for convergence.
    - `init_params`: Method to initialize the weights, means, and precisions. (e.g. “kmeans”, “random”, etc.)
  - **Pro**: Flexible and fast, captures clusters with different shapes and sizes.
  - **Con**: Requires more computational power, sensitive to initialization, and can diverge with too few datapoints to a component.

[13]:
```
gidx = np.array([gauss_list[i]['scores'] for i in range(len(gauss_list))]).
    ↪argmax()
gdata = all_my_exes[gauss_list[gidx]['data']]
make_plots(GaussianMixture, gauss_list[gidx]['params'], gdata, 'Full Dimensions␣
    ↪Unit Variance Scaled')
gauss_list[gidx]
```

[13]:
```
{'params': {'n_components': 2,
   'covariance_type': 'tied',
   'init_params': 'kmeans',
   'n_init': 1,
   'random_state': 42},
  'scores': 0.21646534987980354,
  'time': 0.008425235748291016,
  'model': 'GaussianMixture',
  'data': 'X_std'}
```





Unfortunately, gaussian mixture proved unable to identify closer to the ground truth at well. It constructs clusters which identify 'sex' values only. My hope was that gaussian mixture's ability to handle anisotropic shapes and potential overlap would afford it better flexibility in the face of our difficult data, but unfortunately it proved ineffective. The result of 'tied' implies that clusters should have roughly the same variance and correlation structure among features, which makes sense given out visualization. Given that these results match with the outcome in k-means and that k-means has been selected for as the method of initialization in this case, it is clear that the technique of improving assignments by optimizing for the expectation produced by them in a component could not bare greater results where we focused on a metric like v-measure.

0.1.15 Spectral Clustering

Spectral Clustering uses the eigenvalues of a similarity matrix (e.g. the normalized Laplacian of the adjacency graph) to perform dimensionality reduction before (usually) applying k-means. This makes it effective for clusters that are connected but may not be well-separated in Euclidean space. Spectral clustering is useful when a measure of the center and spread of the cluster is not a suitable description of the complete cluster, such as when clusters are nested circles on the 2D plane.

In Spectral Clustering, the algorithm initially uses an interpretation of the data as a graph from the table of data points, where each data point becomes a node in the graph, and edges between nodes represent pairwise similarities or "affinities" between data points. This similarity graph's edges are stored as a similarity matrix (or adjacency matrix), the values of which form weights used to construct the Laplacian used for eigenvalue decomposition. Using the k eigenvectors of the decomposition of this Laplacian to map the n original datapoints, called "spectral embedding", is followed by clustering for k components.

The process is summarized as: 1. Construct a matrix representation of the graph 2. Compute first k eigenvalues/eigenvectors of this matrix to map to lower dimension 3. Assign points via selected scheme, such as k-means

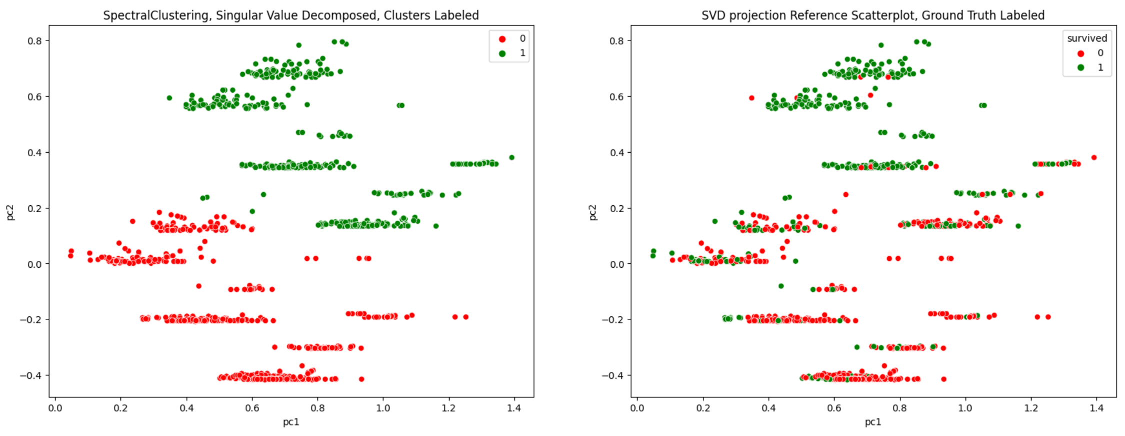
At the time when SpectralClustering is fitted, an affinity matrix is constructed using either a kernel function such the RBF kernel with Euclidean distance, or a k-nearest-neighbors connectivity matrix. However, other kernels reminiscent of Support Vector Machines are useable, i.e. 'polynomial', 'sigmoid', 'cosine', etc.

- **Some Particulars:**
  - Hyperparameters:
    * `n_clusters`: Number of clusters.
    * `affinity`: Method for calculating the similarity matrix ("nearest_neighbors", "rbf", other pairwise kernels).
    * `n_neighbors`: Used when `affinity` is "nearest_neighbors" to construct the graph.
    * `assign_labels`: Strategy for assigning labels in the embedding space. ("kmeans", "cluster_qr", etc.)
  - **Pro**: Handles complex cluster structures, especially in non-convex or manifold data.
  - **Con**: Computationally expensive, especially for large similarity matrices.

[14]:
```
sidx = np.array([spectral_list[i]['scores'] for i in␣
 ↪range(len(spectral_list))]).argmax()
```

```
sdata = all_my_exes[spectral_list[sidx]['data']]
make_plots(SpectralClustering, spectral_list[sidx]['params'], sdata, 'Singular␣
 ↪Value Decomposed')
spectral_list[sidx]
```

```
[14]: {'params': {'n_clusters': 2,
        'affinity': 'rbf',
        'n_init': 3,
        'gamma': 4,
        'assign_labels': 'kmeans',
        'n_jobs': -1,
        'random_state': 42},
       'scores': 0.21676123665253896,
       'time': 1.4019050598144531,
       'model': 'SpectralClustering',
       'data': 'X_svd'}
```



Spectral is a fascinating case because it can actually perform dimensionality reduction in its process via matrix factorization, and so conceptually ties into the work of the project. It is however, a very complicated and advanced technique. Because the number of clusters is two in this case, spectral clustering is cutting the graph in two so that the weight of the edges cut is small compared to the weights of the edges inside each cluster. We can also recognize that because spectral is conducting it's own eigenvalue decomposition, that the results of different dataset versions (different dimensions or projections, as seen in the prior elbow plot) perform identically. Spectral clustering is credited with performing well on non-convex (not circular or eliptically shaped) clusters or when boundaries are not linearly separable. This obviously has limits. As the ground truth shows, boundaries between classes not only overlap but also spread out across the space. This proves too much for the method.

```
[15]: def bench_clst(clst, params, data, labels):
          """Benchmark to evaluate clustering algorithms.
```

```
    Parameters
    ----------
    clst : clustering instance
        A :class:`~sklearn.cluster.???` instance with the initialization
        already set.
    params : dict
        parameters to be used with clustering instance
    data : ndarray of shape (n_samples, n_features)
        The data to cluster.
    labels : ndarray of shape (n_samples,)
        The labels used to compute the clustering metrics which requires some
        supervision.
    """
    t0 = time.time()
    estimator = clst(**params).fit(data)
    fit_time = time.time() - t0
    results = [clst.__name__[:6], fit_time]

    # Define the metrics which require only the true labels and estimator
    # labels
    clustering_metrics = [
        homogeneity_score,
        completeness_score,
        v_measure_score,
        adjusted_rand_score,
        adjusted_mutual_info_score,
    ]

    if clst.__name__ == "GaussianMixture":
        predictions = estimator.predict(data)
        results += [m(labels, predictions) for m in clustering_metrics]
        results += [silhouette_score(data,predictions, metric="euclidean",␣
↪sample_size=300,)]
    else:
        results += [m(labels, estimator.labels_) for m in clustering_metrics]
        # The silhouette score requires the full dataset
        results += [silhouette_score(data,estimator.labels_,␣
↪metric="euclidean", sample_size=300,)]

    # Show the results
    formatter_result = (
        "{:9s}\t{:.3f}s\t{:.3f}\t{:.3f}\t{:.3f}\t{:.3f}\t{:.3f}\t{:.3f}"
    )
    print(formatter_result.format(*results))
    return results
```

```
[16]: print(75 * "_")
      print("init\t\ttime\thomog\tcompl\tv-meas\tARI\tAMI\tsilhouette")

      k = bench_clst(clst=KMeans, params=kmeans_list[kidx]['params'], data=kdata,␣
       ↪labels=gtruth)

      a = bench_clst(clst=AgglomerativeClustering,␣
       ↪params=agglom_list[aidx]['params'], data=adata, labels=gtruth)

      g = bench_clst(clst=GaussianMixture, params=gauss_list[gidx]['params'],␣
       ↪data=gdata, labels=gtruth)

      s = bench_clst(clst=SpectralClustering, params=spectral_list[sidx]['params'],␣
       ↪data=sdata, labels=gtruth)

      print(75 * "_")
```

```
___________________________________________________________________________
init            time    homog   compl   v-meas  ARI     AMI     silhouette
KMeans          0.007s  0.214   0.219   0.216   0.310   0.216   0.366
Agglom          0.025s  0.235   0.285   0.257   0.325   0.257   0.451
Gaussi          0.011s  0.214   0.219   0.216   0.310   0.216   0.245
Spectr          1.102s  0.215   0.219   0.217   0.310   0.216   0.513
___________________________________________________________________________
```

0.1.16 Clustering Metrics

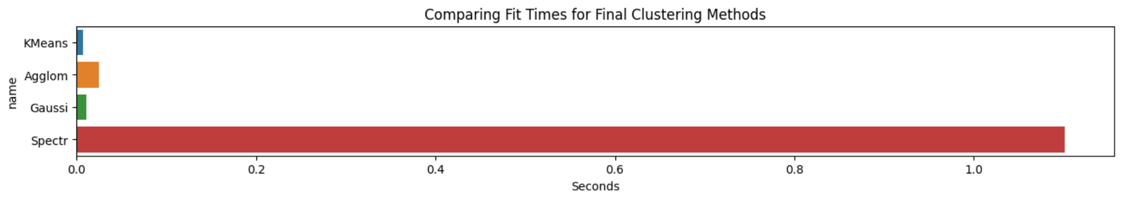
Clustering metrics can vary significantly from those we have seen with supervised methods and amongst one another. Learning about different ones also introduces the concept of using multiple metrics to form consensus. This sklearn clustering metric tutorial was very helpful for contructing a collective benchmark of the performance of a given clustering model. I decided to then use the metrics its original version (specific to k-means) incorporated in order to examine and compare between model's performance. These metrics are used for final comparisons, with some serving unique purposes and others working toward consensus:

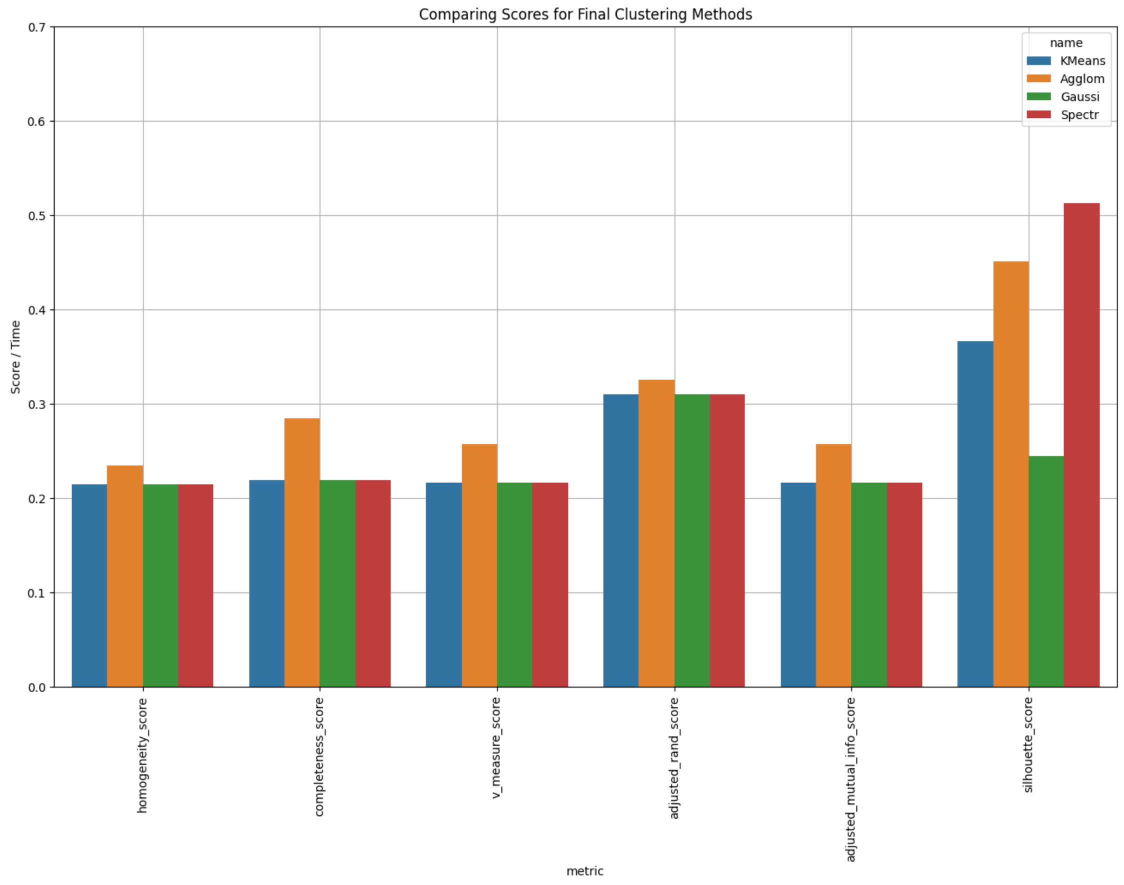
- **Homogeneity Score**: Measures if each cluster contains only members of a single class. Higher values indicate cleaner class splits.
- **Completeness Score**: Measures if all members of a given class are assigned to the same cluster. Completeness complements homogeneity.
- **V-Measure Score**: The harmonic mean of homogeneity and completeness, balancing the two.
- **Adjusted Rand Score**: Measures similarity between two clustering results, adjusted for chance. Useful for comparing clustering results when ground truth is known.
- **Adjusted Mutual Information Score**: A normalization of the Mutual Information score that adjusts for chance, indicating the agreement of two clusterings.
- **Silhouette Score**: Measures how similar an object is to its own cluster compared to other clusters, ranging from -1 to 1. The score is an averaging across the observations.
- **Fit Time**: The time in seconds for the algorithm to assign clusters for the data paired with

it.

KMeans also features the metric “inertia”, which is a measure of how internally coherent the clusters are, and is defined as the sum of squared distances between each point and the centroid of the cluster to which it belongs. This is also the loss function used by k-means at the evaluation stage after it performs cluster reassignment. Lower inertia would mean tighter clusters (though not necessarily better clustering). However, inertia is specific to algorithms that use centroids and squared Euclidean distance, and would not be an option to compare between these four clustering algorithms. Likewise, GaussianMixture offers the log-likelihood for each datapoint, which could be summed to construct a broad metric to use for gaussian mixture model to model comparison, but would not be available across different algorithms.

[17]:
```
metrics = ['fit_time_seconds',
           'homogeneity_score',
           'completeness_score',
           'v_measure_score',
           'adjusted_rand_score',
           'adjusted_mutual_info_score',
           'silhouette_score']
results = pd.DataFrame([k,a,g,s], columns=['name']+metrics)
ftimes = results.loc[:,['name','fit_time_seconds']]
melt = results.drop('fit_time_seconds', axis=1).melt(id_vars='name',␣
 ↪var_name='metric', value_name='values')

plt.figure(figsize=(16, 2))
sns.barplot(data=ftimes, x='fit_time_seconds', y='name', orient='h')
plt.title("Comparing Fit Times for Final Clustering Methods")
plt.xlabel("Seconds")
plt.show()

plt.figure(figsize=(16, 10))
ax = sns.barplot(data=melt, x='metric', y='values', hue='name', zorder=2)
plt.grid(True,zorder=0)
plt.xticks(rotation=90)
ax.set_ylim(0, .7)
ax.set_title("Comparing Scores for Final Clustering Methods")
ax.set_ylabel("Score / Time")
plt.show()
```





0.1.17 Comparing Outcomes

We can now take the opportunity to examine where we have wound up between different algorithms and the feature transformations which we found worked better for them. This means examining a collection of metrics. Using a consensus or combination of related metrics gives a broader perspective on clustering similarity. This helps prevent bias from any one measure's limitations, offering a more reliable assessment.

As was seen earlier, overall purity is bad and hence the homogeneity score is also low - a perfect score would be 1. Completeness relates to homogeneity, but is about cross-over between classes. These two metrics are in alignment with one another. This similarity is itself telling and leads to v-measure, a harmonic mean of the two previous. because these metrics are virtually identical in this case, all three can feel redundant but using all of them can be helpful for interpreting any trade-off between homogeneity and completeness. this is reminiscent of ROC, which attempts to observe change between true positivity and false positivity. The low values here show that some improvement has been achieved but that the results are overall weak for features present. For these particular metrics Kmeans, gaussian, and spectral all stand identically with agglomerative winning out.

Adjusted rand score (ranging anywhere from -0.5 to 1) is comparing between all pairs of ground truth labels and their observation's prediction for each algorithm, and are most aligned between our metric choices. It does not repesent cluster structure, but quantifies the amount of true positives and true negatives. Although it does not really contribute in this case to discerning a choice of best algorithm, it helps to illustrate that some minimal performance has been accomplished. Adjusted mutual information measures similarity between the ground truth and the clustering, adjusting for chance. It is fundamentally based on the intersection and cardinality of the assignments compared. Because it is very similar to another metric, normalized mutual information score, in this case we see that it happens to be identical to v-measure values because v-measure and normalized information score are numerically equivalent for the default settings of NMI (conceptually, homogeneity and completeness are also derived from mutual information). The same as v-measure, we see that agglomerative has slight improvement with all others equal.

Thankfully, we see some differentiation with the remaining metrics. Sillouhette score, as a measure of how similar an observation is to its own cluster compared to other clusters, ranges between -1 and 1. We can see that spectral clustering wins out with agglomerative noticeably behind and k-means in third. Spectral clustering still only has middling performance for sillhouette score. However, this seems likely to do with our choice of lower dimension data to pair it with. Because feature reduction by SVD has not led to reduced accuracy perfromance with agglomerative and spectral, we may be seeing better discernability between observations by the used distance metric of silhouette on average. This is, at least, further reason to conclude that dimension reduction does not necessarily guarantee worse performance under all metrics. Finally, the most obvious distinction between models is their required fit time. Kmeans is clearly our winner in terms of speed, and with agglomerative clustering and gaussian mixture still performing fairly quickly. Spectral analysis presents its greatest drawback here. In practice, spectral takes anywhere from one to several seconds to fit the data. Although it is an interesting algorithm to include for the sake of the project, this difference provides strong reasoning to exclude its use broadly.

After exploring clustering evaluation metrics further, it seemed it might be possible to improve the performance of the models found from hyperparameter searching by experimenting with these metrics; by using different ones among them as the estimator function for grid search. Unfortunately, when returning to the benchmark here outcomes remained consistent - except in the case of silhouette score, which varied regularly from this.

0.1.18 Wrapping Up

In the end we have poor clustering performance resulting from the lack of separability in the data. Typically this might mean going back again for even more EDA and perhaps sourcing data differently, but in this case we are very unlikely to find further variables for these observations. As I would expect occurs for many machine learning problems, attempting to apply the Titanic data in this way amounts to a gamble which did not pay off. Ultimately our lack of performance cannot be attributed to clustering itself, and not necessarily to the various preprocessing and projection techniques we explored. Instead it seems reasonable to say that we have run into limitations with what the available features of the data are able to provide. Although it remains possible that among feature engineering techniques there exists a better suited method, or methods for the applications we were attempting. Improvement with the right variables (where they actually exist) and further exposure to engineering techniques is certainly possible. Critically though, I am ready at this point to label the best route for improvement to be to reassess which kind of machine learning problem we wish to apply this data.

Despite the shortcomings, the project was able to explore the effects of matrix factorization and projection in a fairly interpretable way. We saw how the results can be visually distinguished for the range of values available in features, with more continuous variability from numeric features and separable, isolated groupings from categorical features. We saw that preprocessing and feature engineering could produce a wide and difficult to predict array of results with the data, and control how it is organized in the feature space. Importantly, there is no single fixed orientation between observations where we consider the results of the preprocessing/engineering techniques explored. It did not work well in this case, but we have seen that variability of features can be (potentially) aligned in concert with one another to construct useful orientations within a feature space, and where this is managed consistently enough will effectively resemble an orientation which the inclusion of our ground truth might provide. In terms of unsupervised clustering with no predetermined ground truth we are then able to recognize that where some (fairly) separable, distinct groupings can be found which are not themselves merely consistent with one or two directly identifiable features but a latent result of many variables functioning in concert, then these groupings are successfully identifying a perhaps yet undefind relationship among their observations.

0.1.19 References

1. Kaggle: Paiva, V. (n.d.). *The Complete Titanic Dataset (Version 3)*. Kaggle. Retrieved from `https://www.kaggle.com/datasets/vinicius150987/titanic3`
2. Wikipedia. (n.d.). *Passengers of the Titanic*. Wikipedia. Retrieved from `https://en.wikipedia.org/wiki/Passengers_of_the_Titanic`
3. Wikipedia. (n.d.). *Singular value decomposition*. Wikipedia. Retrieved from `https://en.wikipedia.org/wiki/Singular_value_decomposition`
4. Ullman, J. D., & Rajaraman, A. (n.d.). *Mining of massive datasets, Chapter 11: Dimensionality reduction*. Stanford University. Retrieved from `http://infolab.stanford.edu/~ullman/mmds/ch11.pdf`
5. Park, H. (Mar 13, 2016). *Nonnegative Matrix Factorizations for Clustering, Haesun Park, Georgia Institute of Technology* [Video]. YouTube. Retrieved from `https://www.youtube.com/watch?v=EKvh4ANUHWM`
6. scikit-learn. (n.d.). *MinMaxScaler documentation*. Retrieved from `https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.MinMaxScaler.html`
7. scikit-learn. (n.d.). *StandardScaler documentation*. Retrieved from `https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.StandardScaler.html`
8. scikit-learn. (n.d.). *Non-negative matrix factorization (NMF) documentation*. Retrieved from `https://scikit-learn.org/stable/modules/generated/sklearn.decomposition.NMF.html`
9. scikit-learn. (n.d.). *Truncated singular value decomposition (TruncatedSVD) documentation*. Retrieved from `https://scikit-learn.org/stable/modules/generated/sklearn.decomposition.TruncatedSVD.html`
10. scikit-learn. (n.d.). *Clustering documentation*. Retrieved from `https://scikit-learn.org/stable/modules/clustering.html`
11. scikit-learn. (n.d.). *Comparing different clustering algorithms on toy datasets*. Retrieved from `https://scikit-learn.org/stable/auto_examples/cluster/plot_cluster_`

comparison.html

12. scikit-learn. (n.d.). *AgglomerativeClustering documentation*. Retrieved from https://scikit-learn.org/stable/modules/generated/sklearn.cluster.AgglomerativeClustering.html
13. scikit-learn. (n.d.). *KMeans documentation*. Retrieved from https://scikit-learn.org/stable/modules/generated/sklearn.cluster.KMeans.html
14. scikit-learn. (n.d.). *GaussianMixture documentation*. Retrieved from https://scikit-learn.org/stable/modules/generated/sklearn.mixture.GaussianMixture.html
15. scikit-learn. (n.d.). *SpectralClustering documentation*. Retrieved from https://scikit-learn.org/stable/modules/generated/sklearn.cluster.SpectralClustering.html
16. scikit-learn. (n.d.). *A demo of K-Means clustering on the handwritten digits data*. Retrieved from https://scikit-learn.org/stable/auto_examples/cluster/plot_kmeans_digits.html

```
[ ]:
```