MedicalProject

December 6, 2020

In this project, we will observe data on $10,000$ patients to determine how likely an active treatment is to cause a recovery.

For each patient, we have attributes x where: - $x_1 \in \{0,1\}$, sex - $x_2 \in \{0,1\}$, smoker - $x_{3:128} \in \{0,1\}^{125}$, gene expressions (possibly missing values) - $x_{129:130} \in \{0,1\}^2$, symptoms

Each pateint also recieved a therapeutic intervention $a \in A$ which was followed by their recovery outcome $y \in \{0,1\}$.

1 Reading data

We can start the analysis by reading in the data. We will also import our needed libraries.

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

[2]:
```
# autoreload local objects to avoid kernel restarts
%load_ext autoreload
%autoreload 2
import warnings
warnings.filterwarnings('ignore')
```

[3]:
```
people = ['gender', 'smoker']
genes = ['gene'+str(i+1) for i in range(126)]
symptoms = ['symptom1', 'symptom2']
```

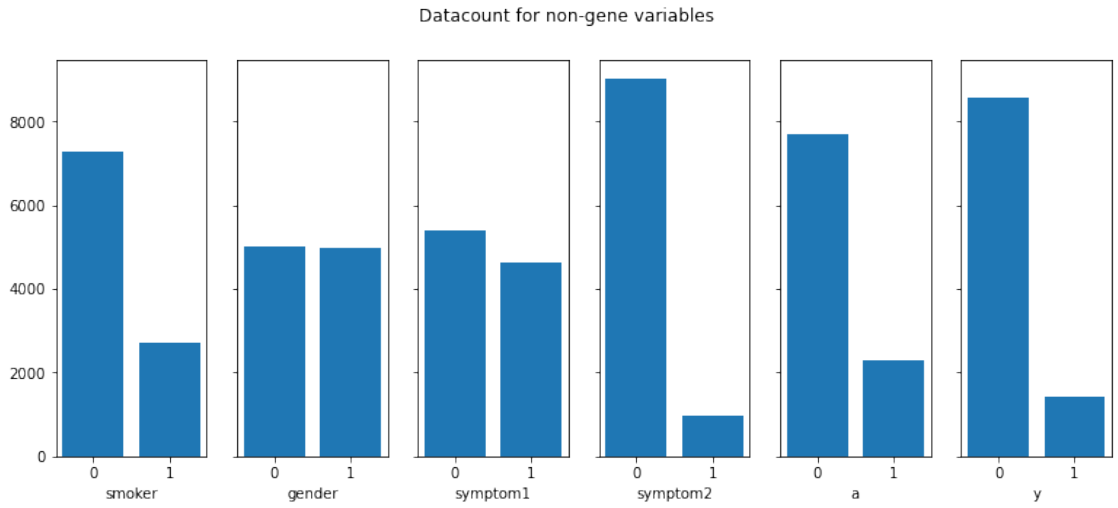
[4]:
```
def read_data(sample_size=None):
    colnames = people + genes + symptoms
    dfx = pd.read_table("data/historical_X.dat", sep=" ", header=None,␣
 ↪names=colnames)
    dfa = pd.read_table("data/historical_A.dat", sep=" ", header=None,␣
 ↪names=['a'])
    dfy = pd.read_table("data/historical_Y.dat", sep=" ", header=None,␣
 ↪names=['y'])
    df = pd.concat([dfx,dfa,dfy],axis=1) #combining data to one dataframe
    if sample_size:
```

```
        df = df.sample(sample_size)
        df.index = range(sample_size)
    return df, dfx, dfa, dfy, colnames
```

[5]:
```
df, X, a, y, columns = read_data()
```

We can get a quick overview of teh distributions of some of the features, to see how balanced our data set is.

[6]:
```
non_genes = ["smoker","gender", "symptom1", "symptom2", "a","y"]
fig,ax = plt.subplots(nrows=1, ncols=6, sharey=True, figsize=(13,5))
for i,feature in enumerate(non_genes):
    ax[i].bar(x=[0,1],height=df[feature].value_counts())
    ax[i].set_xlabel(feature)
fig.suptitle("Datacount for non-gene variables")
plt.show()
```



The data is really only balance over gender (of the features shown above). The unbalance of the other features must be taken into consider when looking at distributions later. (In a more thorough analysis, synthetic data could be generated to balance these out, using techniques like SMOTE. However, these approaches are outside the scope of this assignment.)

We now need to make sure to seperate our data so that we can evalute on completely new data when our model has been built and tuned. We can set a seed so that the exact results can be reproduced.

[7]:
```
from sklearn.model_selection import train_test_split
import random
random.seed(6122020)
```

[8]:
```
# train test split of the data
dftrain, dftest, _, _ = train_test_split(df, y, test_size=0.25)
# change this into sample()
```

[85]:
```
Xtr, Xte = dftrain.drop(['a','y'], axis=1), dftest.drop(['a','y'], axis=1)
atr, ate = dftrain.a, dftest.a
ytr, yte = dftrain.y, dftest.y
```

[10]:
```
Xtr.head()
```

[10]:

	gender	smoker	gene1	gene2	gene3	gene4	gene5	gene6	gene7	gene8	\
5064	0	1	1	1	0	0	1	1	0	0	
5634	0	0	0	1	1	1	0	0	0	0	
6425	1	0	0	0	0	1	1	0	1	1	
336	0	0	1	1	1	1	1	1	0	1	
7433	0	1	1	0	1	0	1	1	0	1	

	…	gene119	gene120	gene121	gene122	gene123	gene124	gene125	\
5064	…	0	0	0	1	0	0	1	
5634	…	1	0	1	0	0	1	0	
6425	…	0	1	0	1	1	0	0	
336	…	1	0	1	0	1	1	0	
7433	…	1	0	1	0	1	1	0	

	gene126	symptom1	symptom2
5064	0	1	0
5634	0	1	0
6425	1	0	0
336	1	1	0
7433	1	0	0

[5 rows x 130 columns]

[11]:
```
atr.head()
```

[11]:
```
5064    0
5634    0
6425    0
336     1
7433    0
Name: a, dtype: int64
```

2 Structure in data

It is uncertian if the symptoms are due to the same disease or caused by different conditions that give similar symptoms. We will try to estimate whether a single-cause model can represent the data, or if we need to assume multiple causes. This can be done by using a clustering algorithm

on our features. If we get multiple clusters, then it is likely that there are multiple causes of our registered symptoms.

Before we start clustering though, we'd like to reduce our dimensions, so we don't waste time crunching too many unnecessary variables. This will be done using sklearn's `PCA` method. We can define an arbitrary number of components `n_components=10` to tell the method how many compents we want our data reduced to.

2.1 Dimension reduction

Principal component analysis finds the features the contribute most to the variance of the data, and projects the values of the remaining features onto this axis. This is done for every datapoint, meaning each component will be an array of length $10,000$. We'll need to take the mean of of each component to get the average pricapal compoents for our data.

[12]:
```
from sklearn.decomposition import PCA

genes = columns[2:-2]
df_genes = df[genes]

pca = PCA(n_components=10)
pca.fit(df_genes)
pca_vals = pca.transform(df_genes)
abs(pca_vals).mean(axis=0)
```

[12]:
```
array([1.69475384, 1.5224524 , 1.44626267, 1.38638171, 1.28197789,
       1.22637358, 1.15590674, 1.09537984, 0.43629035, 0.32586388])
```

[13]:
```
pca_vals.shape
```

[13]:
```
(10000, 10)
```

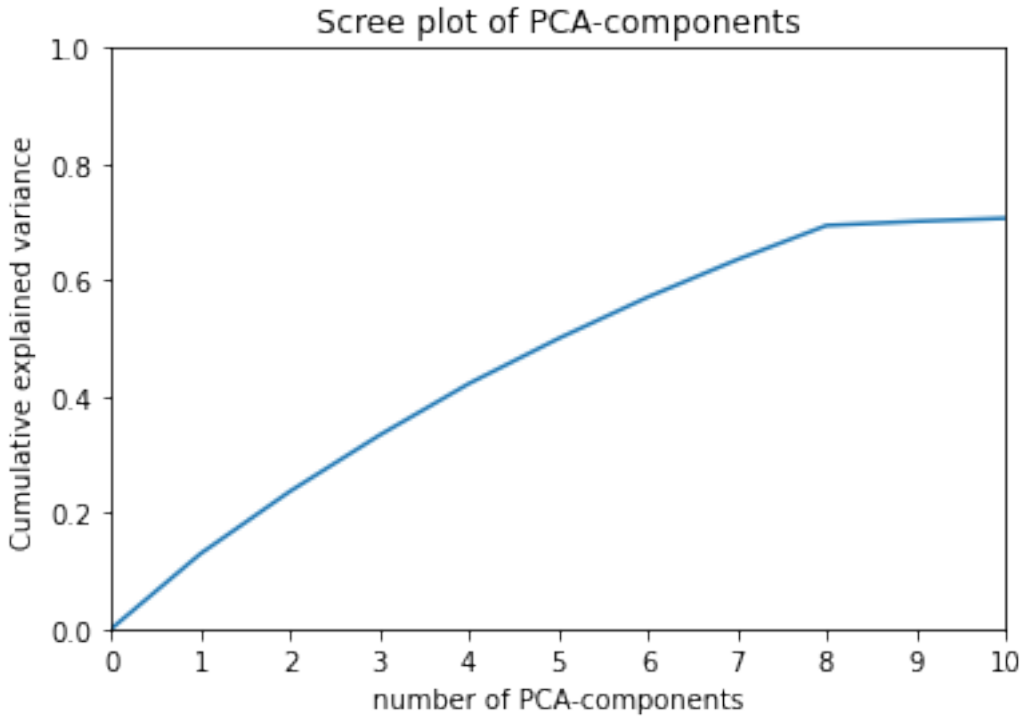
The sudden drop in PCA values between component 8 and 9 indicates that 8 principal components explain most of the varaince within the data. Before we continue using only these 8 components, let's see how much of the varaince these components are responsible for.

The values for each PCA-component is calles z-scores. Mathematically we describe the z-score for PC1 as: $z_{i1} = \alpha_1 x_{i1} + \alpha_2 x_{i2} + ... + \alpha_p x_{ip}$.

We can create a sree-plot which describes the cummulative proportion of variance explained.

[14]:
```
y = (list(np.cumsum(pca.explained_variance_ratio_)))
y.insert(0,0)
x = range(11)
plt.plot(x,y)
plt.xlabel("number of PCA-components")
plt.ylabel("Cumulative explained variance")
plt.ylim([0,1])
plt.xlim([0,10])
plt.xticks([0,1,2,3,4,5,6,7,8,9,10])
```

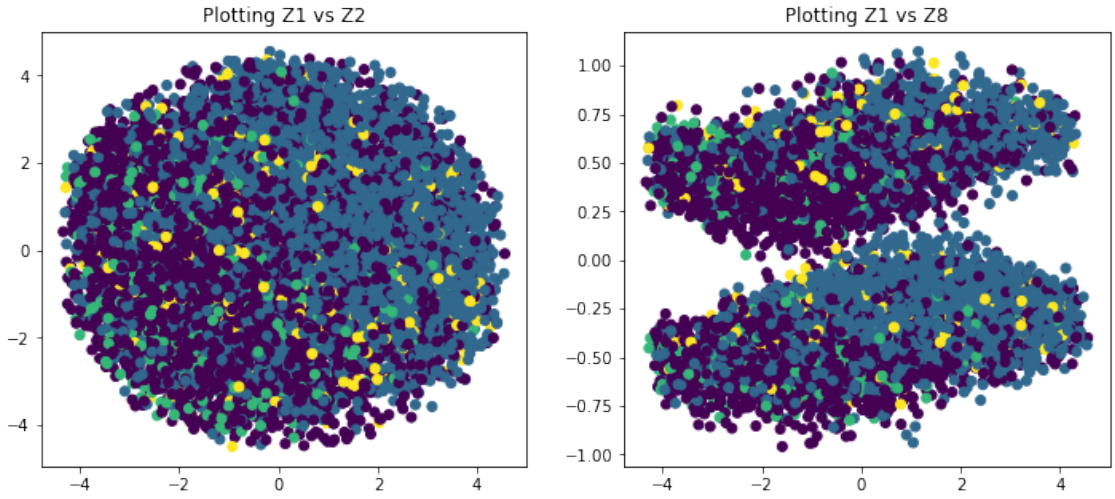
```
plt.title("Scree plot of PCA-components")
plt.show()
```



Here, we will see that 8 principal components describe approximately 70% of the data, as mentioned numerically above. These 8 components can then be used as a "surrogate" for the 126 variables.

Plotting different combinations of principal components we see that most of the plots just describe a big blur, but with a kind of clustering in different symptom values (colors). Interestingly enough, when plotting PC1 vs PC8 we see 2 distinct clusters in the gene expression data. Coloring points according to the 4 different symptom classes we can have, (0,0), (0, 1), (1, 0), (1, 1). The results indicate that different causes can give the same symptoms, hence that there could be people with **different diseases** in the dataset.

[15]:
```
s1s2 = df['symptom1']*2**0 + df['symptom2']*2**1

fig, (ax1,ax2) = plt.subplots(nrows=1, ncols=2, figsize=(12,5))
ax1.scatter(pca_vals[:,0], pca_vals[:,1], c=s1s2)
ax1.set_title("Plotting Z1 vs Z2")
ax2.scatter(pca_vals[:,0], pca_vals[:,8], c =s1s2)
ax2.set_title("Plotting Z1 vs Z8")
plt.show()
```



It is important to note the "could" in the statement above. This clustering could also indicate some other unmeasured difference between the data points, for instance related to different gene-expressions for different gender.

2.2 Clustering for indications of multiple diseases

To investigate if the symptoms present are all due to the same disease, or if they are different conditions with similar symptoms, we also try to cluster the data with Hierarchical Agglomerative clustering.

[16]:
```
from sklearn.cluster import AgglomerativeClustering
from scipy.cluster.hierarchy import dendrogram, ward
```

[17]:
```
g_names = [f"pca_gene{i}" for i in range(1,9)]
df_pca = pd.DataFrame(pca_vals[:,0:8], columns = g_names)

df_condition = pd.concat([df[people],df_pca], axis=1) #using PC-components␣
↪instead of all genes
df_condition.head()
```

[17]:

	gender	smoker	pca_gene1	pca_gene2	pca_gene3	pca_gene4	pca_gene5	\
0	0	0	2.430039	-0.983775	0.522087	-2.037578	1.780916	
1	0	1	0.634035	1.780100	-1.662433	2.659486	1.965055	
2	0	0	-3.863433	0.788687	-0.224533	-1.146848	0.803919	
3	1	1	-1.953169	2.283368	2.426192	-0.934647	-0.556716	
4	0	1	0.724225	2.297267	-0.441892	2.253432	0.365417	

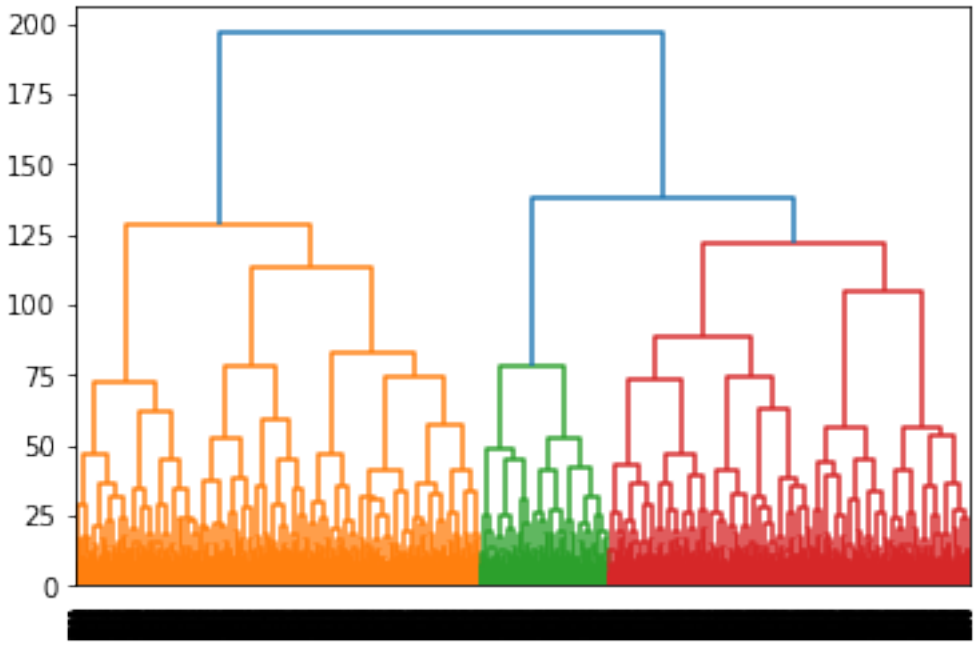
	pca_gene6	pca_gene7	pca_gene8
0	0.708186	2.502221	-0.542676
1	2.220976	-0.151121	-0.614869
2	-1.393783	-1.513266	-1.013838

```
3  -1.765688  -1.133687  -0.094508
4  -2.164467   0.962299  -2.303855
```

[18]:
```
n=4
hac = AgglomerativeClustering(n_clusters=n,linkage="ward")
cluster_labels = hac.fit_predict(df_condition)

clusters = np.unique(cluster_labels, return_counts=True)
print(f"Number of datapoints in each of {n} clusters is: {clusters[1]}\n")
```

```
Number of datapoints in each of 4 clusters is: [4033 3144 1442 1381]
```

[19]:
```
linkage_array = ward(df_condition)
dendrogram(linkage_array)
None
```



From the dendrogram, 3, possibly 4 clusters are evident. HAC does not target a special number of clusters but propose different clusters depending on the data being analyzed.

Let's create a dataframe including only the two symptom varaibles, and the respective cluster labels for our data.

[20]:
```
s = pd.Series(cluster_labels, name="clusters")
df_clusters = pd.concat([df[symptoms], s], axis=1)
df_clusters.head()
```

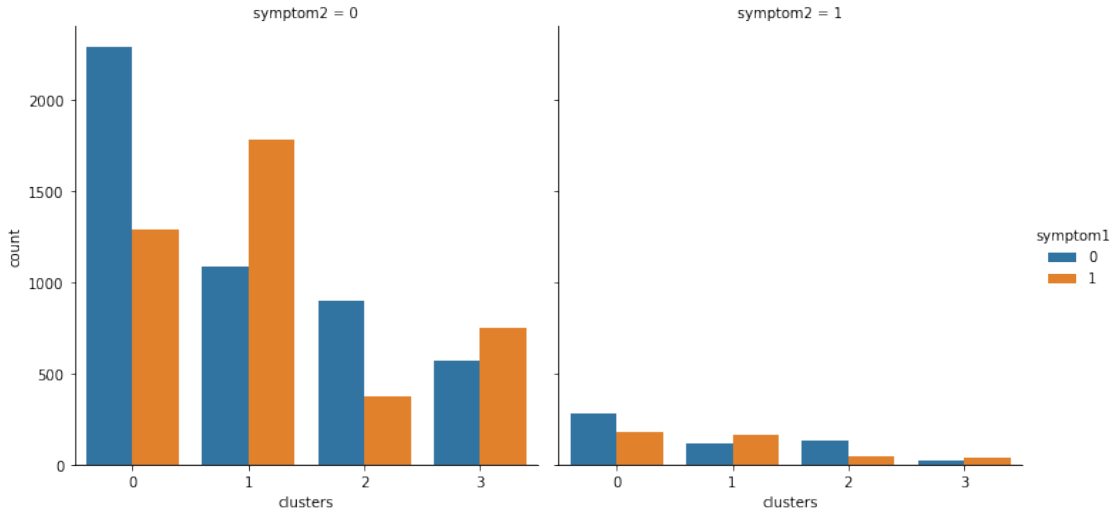
[20]:

	symptom1	symptom2	clusters
0	1	0	1
1	0	0	3
2	0	0	2
3	0	0	2
4	1	0	3

We can then plot the distribution of the different clusters between the two symptoms. This can be done in a combined plot using seaborn's `catplot` method.

[21]:
```
sns.catplot(x="clusters", hue="symptom1", data=df_clusters, kind="count",␣
↪col="symptom2")
```

[21]: <seaborn.axisgrid.FacetGrid at 0x20f66b39760>



Here, the plot on the left represents the instance where symptom 2 is non-exsistent. The coloration of the plots indicates the the prescence of symptom 1 or not.

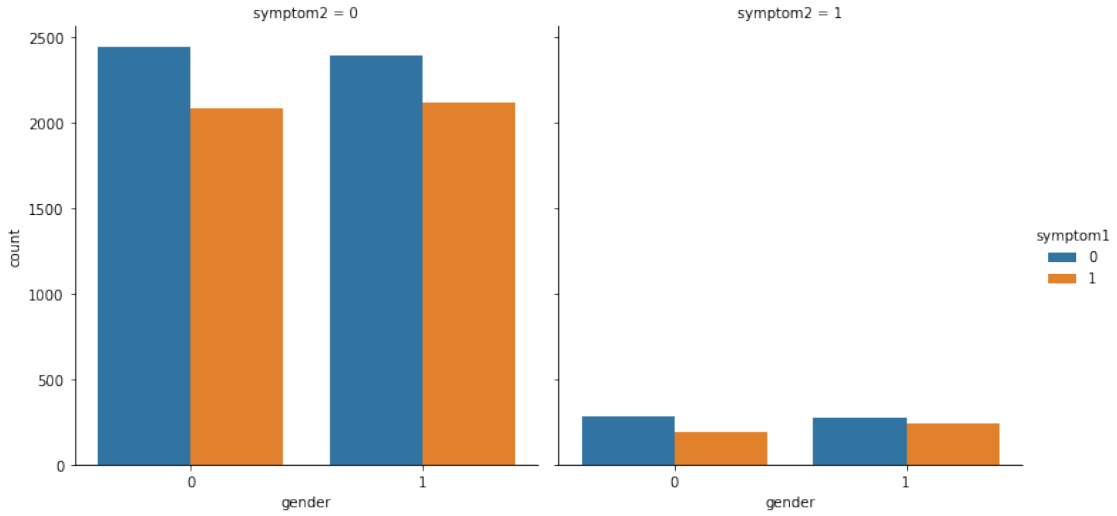
The plots show a diverse distribution between both symptoms, indicating that there can exist **different causes for the same symptoms.**

2.3 Predicting symptoms from single feature

The genes corresponding to the top 8 PCA's can be seen as the most important of the gene data. We can analyze the distributions of symptoms between these genes, gender, and smoking habits, to see whether it is plausible that a single feature can predict the prescence or absence of a symptom.

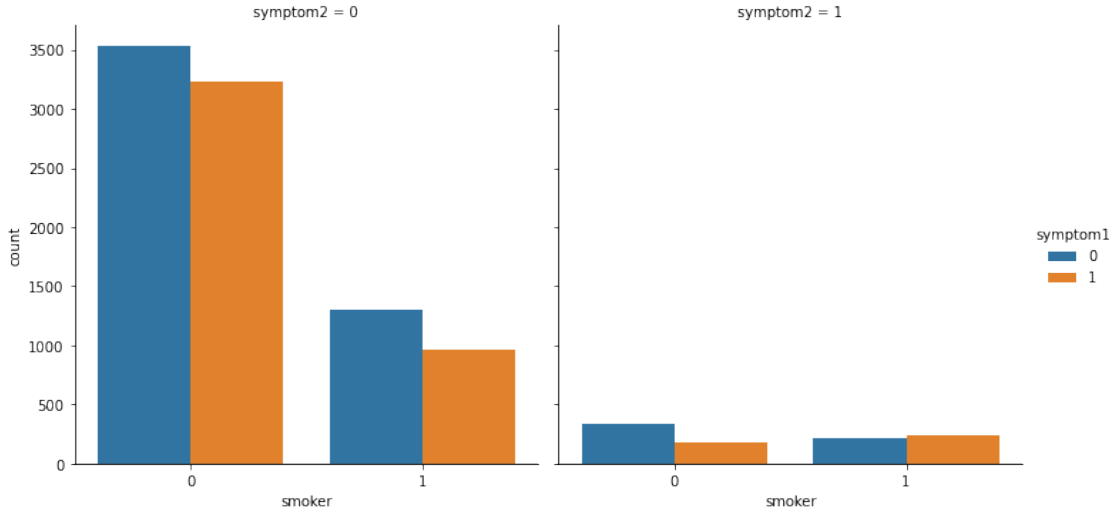
[22]:
```
sns.catplot(x="gender", hue="symptom1", data=df, kind="count", col="symptom2")
```

[22]: <seaborn.axisgrid.FacetGrid at 0x20f09a937f0>



[23]:
```
sns.catplot(x="smoker", hue="symptom1", data=df, kind="count", col="symptom2")
```

[23]: <seaborn.axisgrid.FacetGrid at 0x20f002b8d30>

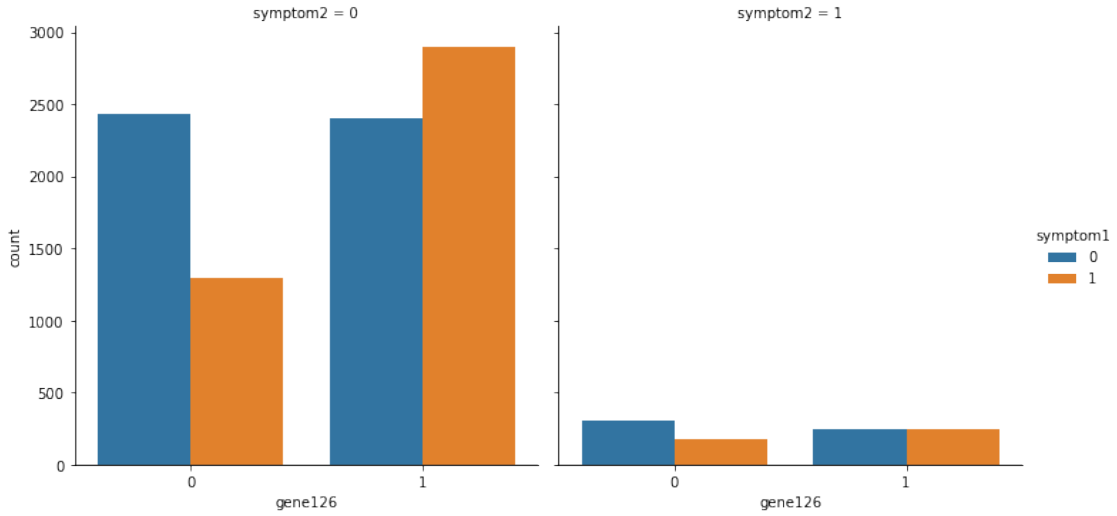


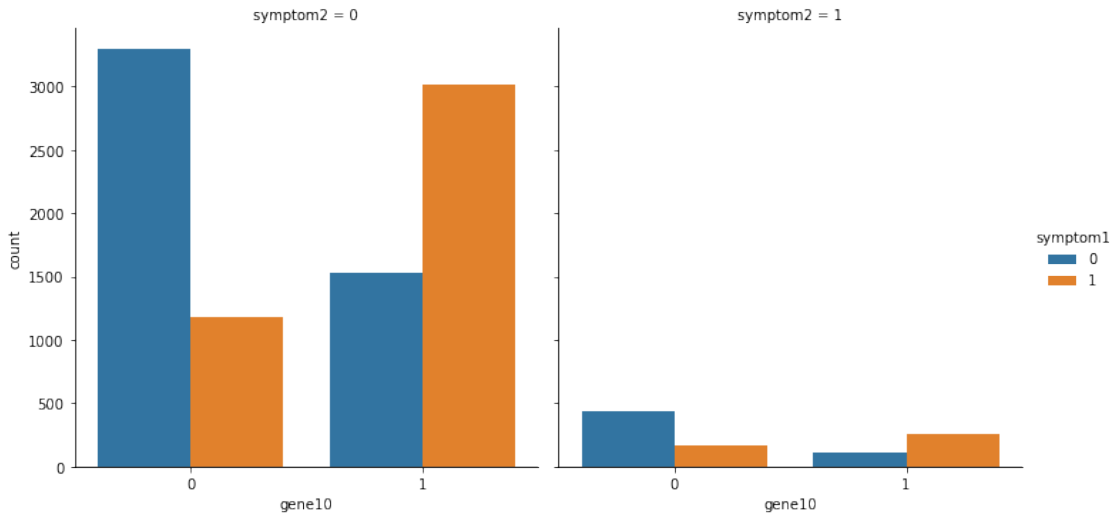
Unfortunately, neither gender nor smoking habits show signs of directly predicitng the prescence of a symptom. This conclusion can be drawn, since the distributions for the two symptoms resemble those from the start of the analysis.

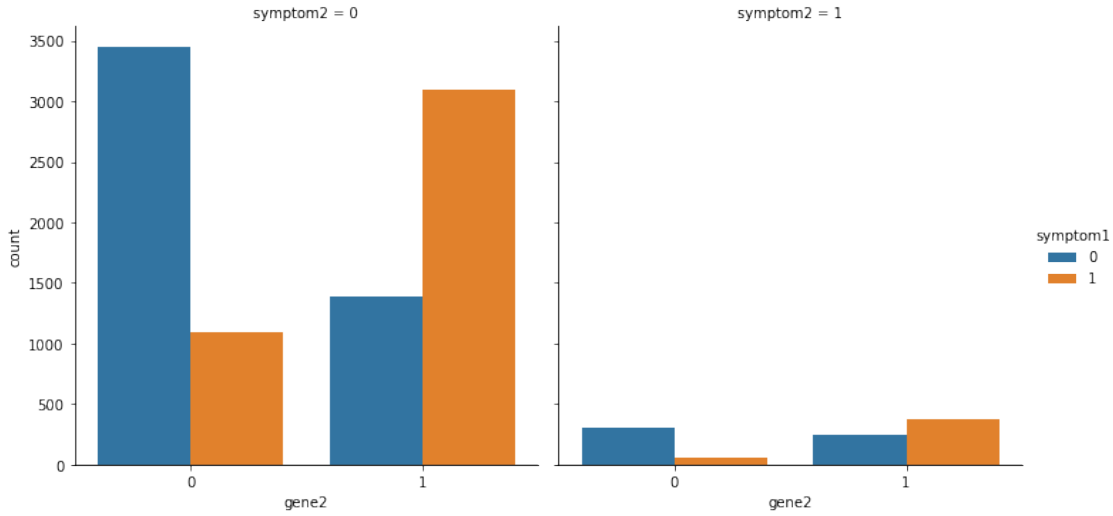
We can try a similar analysis for the top genes. We'll only look at a few here, since most of the plots ended up looking pretty similar.

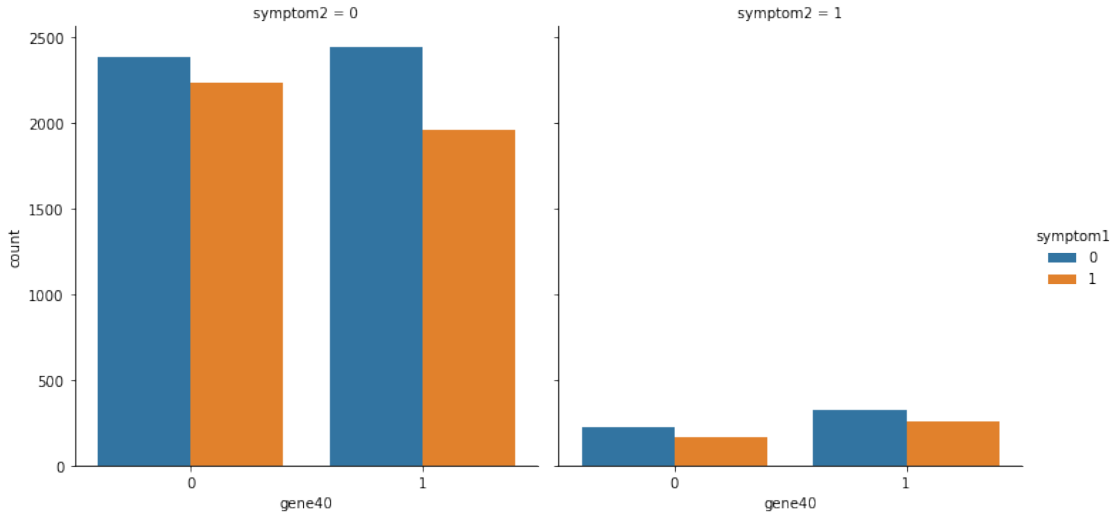
[24]:
```
# rank the genes according to importance from PCA
N = 8  # to limit # of genes to use below
top_genes_idx = np.argsort((-abs(pca.components_).mean(axis=0)))[:N] #read␣
 ↪negated array for descending values
top_genes = ['gene'+str(g+1) for g in top_genes_idx] # we started gene labels␣
 ↪with 1
```

[25]:
```
for gene in top_genes[:4]:
    sns.catplot(x=gene, hue="symptom1", data=df, kind="count", col="symptom2")
```









Again, we see distributions similar to the unbalance between the symptoms in the data set. From this, we assume that more than one feature is needed to predict the presence of a symptom.

2.4 Feature selection

While principal component anlysis helps greatly in dimension reduction, it doesn't neccessary always work well as a feature selector. We can instead implement a forward stepwise feature selection to find the most important features. This procedure will also help us rank the importance of gender and smoking habits *among* the genes, not seperate from.

Here, we will start with an empty model, then add features one by one according to which give the best accuracy score on symptom predictions. We will again look for the top 10 features, to compare with the best ones found above.

The accuracy metric we will use here will be sklearn's `cross_val_score`. This method takes in a classifier, and a set of features, and runs a cross validation on the classifier with those features. The feature giving the best score for the current iteration will then be appended to a list of saved featues, and removed from the possible features of the next iterations. When we've reached our desired set of top 10 features, we will exit the iteration-loop.

[26]:
```
# Forward stepwise feature selection using symptom1 as target
from sklearn.tree import DecisionTreeClassifier
from sklearn.model_selection import cross_val_score

causes = X.drop(['symptom1', 'symptom2'], axis=1)
columns = list(causes.columns)
selected_features = []
scores1 = []
N = 10
while len(selected_features) < N:
    best_score = pd.Series([0])
    best_feature = None
    for feature in columns:
        score = cross_val_score(DecisionTreeClassifier(max_depth=20),
                                causes[selected_features + [feature]],␣
↪df['symptom1'])
        if score.mean() > best_score.mean():
            best_feature = feature
            best_score = score
    print(f"{best_feature: >10}: {best_score} ({best_score.mean()})")
    columns.remove(best_feature)
    selected_features.append(best_feature)
    scores1.append(score)
```

```
     gene4: [0.7885 0.794  0.755  0.771  0.785 ] (0.7787)
    gender: [0.7885 0.794  0.755  0.771  0.785 ] (0.7787)
    smoker: [0.7885 0.794  0.755  0.771  0.785 ] (0.7787)
     gene1: [0.7885 0.794  0.755  0.771  0.785 ] (0.7787)
     gene3: [0.7885 0.794  0.755  0.771  0.785 ] (0.7787)
    gene16: [0.7875 0.793  0.756  0.7725 0.7865] (0.7791)
    gene64: [0.7905 0.7925 0.7545 0.7725 0.7885] (0.7797)
    gene74: [0.7895 0.794  0.7525 0.7725 0.789 ] (0.7795)
    gene24: [0.787  0.7925 0.752  0.7725 0.7845] (0.7777000000000001)
    gene31: [0.787  0.7855 0.7565 0.771  0.784 ] (0.7767999999999999)
```

[78]:
```
for z in zip(selected_features, scores1):
    print(z[0], z[1])
```

```
gender [0.5965 0.5945 0.5625 0.602  0.5855]
smoker [0.7885 0.794  0.755  0.771  0.785 ]
gene1 [0.7885 0.794  0.755  0.771  0.785 ]
gene2 [0.7885 0.794  0.755  0.771  0.785 ]
gene3 [0.7885 0.794  0.755  0.771  0.785 ]
```

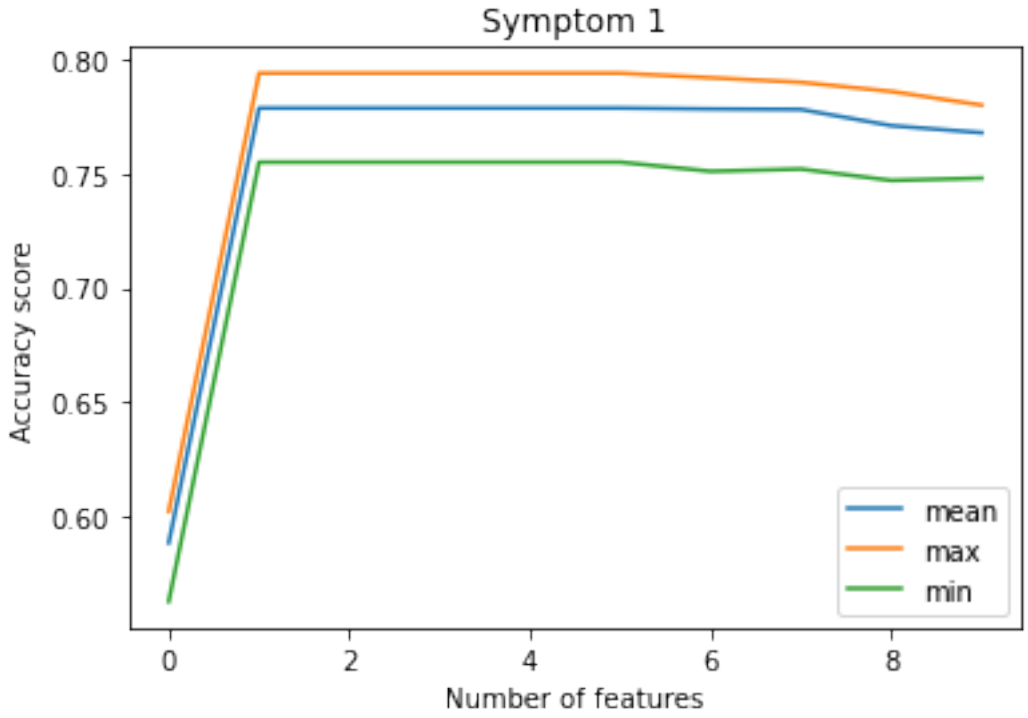
```
gene4 [0.7885 0.794  0.755  0.771  0.785 ]
gene16 [0.787  0.792  0.751  0.774  0.7875]
gene31 [0.7855 0.79   0.752  0.773  0.79  ]
gene80 [0.786  0.784  0.747  0.7595 0.7785]
gene87 [0.7775 0.78   0.748  0.7605 0.7735]
```

[79]:
```
print('abc',
      'cdf')
```

```
abc cdf
```

[27]:
```
plt.plot([s.mean() for s in scores1], label='mean')
plt.plot([s.max() for s in scores1], label='max')
plt.plot([s.min() for s in scores1], label='min')
plt.xlabel('Number of features')
plt.ylabel('Accuracy score')
plt.title('Symptom 1')
plt.legend()
```

[27]:
```
<matplotlib.legend.Legend at 0x20f09aa10a0>
```



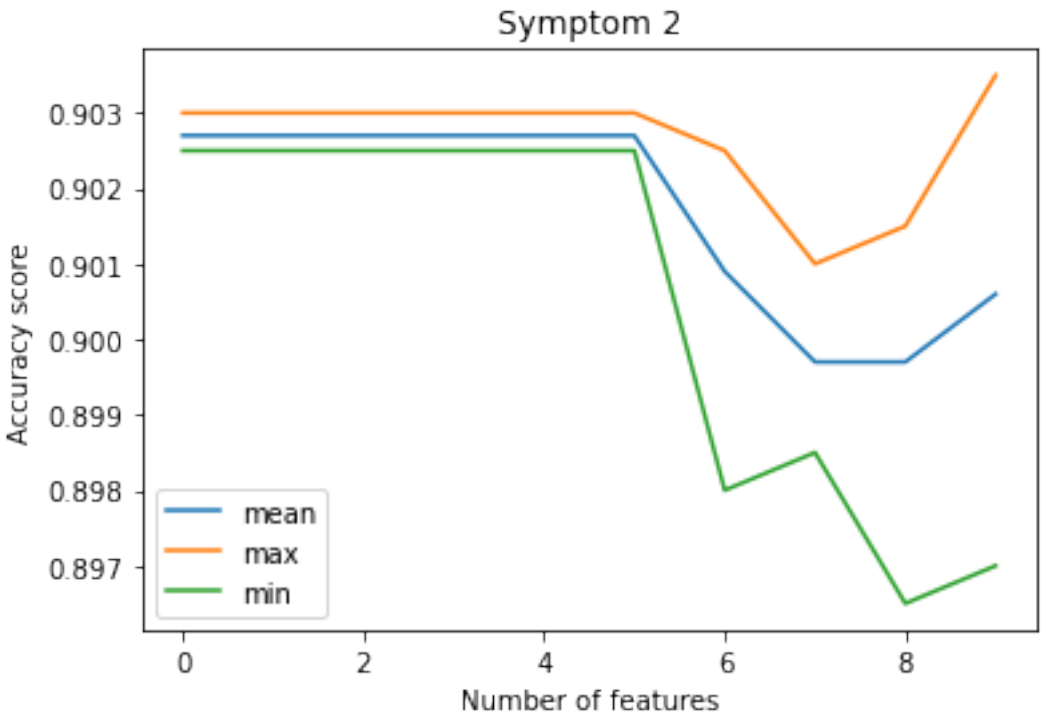
[28]:
```
columns = list(causes.columns)
selected_features = []
scores2 = []
```

```
while len(selected_features) < N:
    best_score = pd.Series([0])
    best_feature = None
    for feature in columns:
        score = cross_val_score(DecisionTreeClassifier(max_depth=20),
                                causes[selected_features + [feature]],␣
↪df['symptom2'])
        if score.mean() > best_score.mean():
            best_feature = feature
            best_score = score
    print(f"{best_feature: >10}: {best_score} ({best_score.mean()})")
    columns.remove(best_feature)
    selected_features.append(best_feature)
    scores2.append(score)
```

```
    gender: [0.903  0.903  0.9025 0.9025 0.9025] (0.9027)
    smoker: [0.903  0.903  0.9025 0.9025 0.9025] (0.9027)
     gene1: [0.903  0.903  0.9025 0.9025 0.9025] (0.9027)
     gene2: [0.903  0.903  0.9025 0.9025 0.9025] (0.9027)
     gene3: [0.903  0.903  0.9025 0.9025 0.9025] (0.9027)
     gene4: [0.903  0.903  0.9025 0.9025 0.9025] (0.9027)
    gene16: [0.9025 0.9035 0.903  0.903  0.903 ] (0.9030000000000001)
    gene31: [0.9035 0.9035 0.9005 0.9035 0.905 ] (0.9032)
    gene80: [0.904  0.9065 0.903  0.9    0.903 ] (0.9033)
    gene87: [0.9025 0.905  0.901  0.8995 0.9045] (0.9024999999999999)
```

[29]:
```
plt.plot([s.mean() for s in scores2], label='mean')
plt.plot([s.max() for s in scores2], label='max')
plt.plot([s.min() for s in scores2], label='min')
plt.xlabel('Number of features')
plt.ylabel('Accuracy score')
plt.title('Symptom 2')
plt.legend()
```

[29]:
```
<matplotlib.legend.Legend at 0x20f0b5cf190>
```



For symptom 1, the average accuracy doesn't change much after the first feature is added. After a few more features are added, we see a slight increase in accuracy followed by a steady decrease. This tells us there does in fact exist some optimal subset of features that could predict this symptom. However, with an accuracy of 70%, it could be argued that predicting this symptom from only these data is not trustworthy enough. In other words, it is hard to conclude that top 5 features are the causes of symptom 1. There must exist some outside factors not measured by the data set.

For symptom 2, we see a flat accuracy as new features are added to the model. When the 6th feature is added, all of the accuracies fall, again indicating some optimal subset of features for predicting symptom 2. It is also important to remember the distribution of symptom 2 found at the begining. There were very few patients with this symptom, making it very hard for a prediction model to make trustworthy estimates for this symptom. Again, we conclude that other data are needed if this symptom is to be modelled correctly (more precisely).

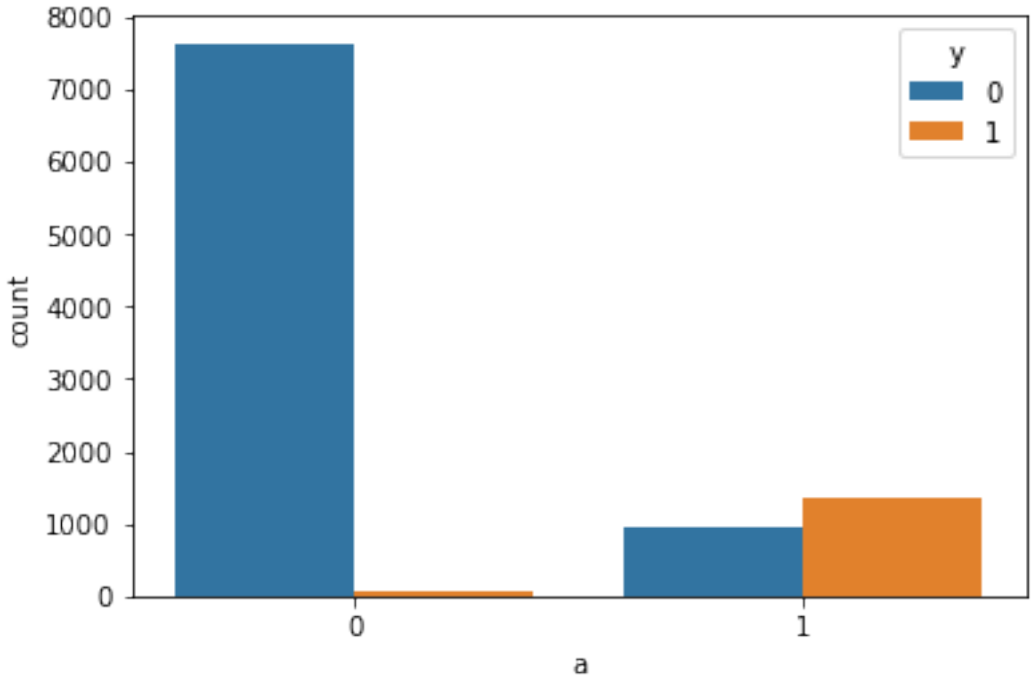
3 Effects of actions

We also observe the effects of two different therapeutic interventions, one of which is placebo $a = 0$, and the other is an experimental active treatment $a = 1$.

We are interested in measuring the effectiveness of the two treatment types to perhaps see if there exists cases where the active treatment is never effective. This will tell us if a simple, fixed policy for always choosing to prescribe active treatment would be desired for these data.

[30]:
```
sns.countplot(x="a", hue="y", data = df)
```

[30]: <matplotlib.axes._subplots.AxesSubplot at 0x20f0b4bda90>



The blue bar on the right shows that there exists cases where the treatment is not effective, i.e. $a = 1$ but $y = 0$. In fact, there are almost as many non-recoveries after active treatment as there are recoveries with the same treatment. Since most medicine unfortunately bear unwanted side-effects, it is *not recommended* to prescribe active treatment to every sick person.

4 Policies

We are now ready to start measuring the total cost-reward trade off for these data. Our goal will be to maximize the total number of patients who recover, while avoiding unnecessary active treatments.

The choices of actions $\{0, 1\}$ in the data we're observing comes from some unknown policy π_0. As previously mentioned, for an individual t, the action value $a_t = 0$ represents those patients given a placebo, while $a_t = 1$ represents those given an active treatment. An outcome value of $y_t = 0$ means patient t did not recover, while $y_t = 1$ means the patient did recover.

We define the utility function for our scenario as:

$$U = \sum_t r_t$$

$$r_t \triangleq -0.1a_t + y_t$$

The definition of the reward r_t says that the effects of the active treatment must be 10% better than the effects of the placebo, hence the negative weight penalizing the actions > 0. (Actions equal to zero will not be affected by the weight.)

Assumption 2.1: *The policies π can be represented as conditional distributions:*

$$\pi(a|x)$$

In other words, policies are defined as the probability distributions of a_t being the correct choice of action given individual data x_t, for all individuals in the dataset, $t \in [1, 10000]$.

4.1 Measuring utility

We now want to measure how well the unknown policy π_0 was at deciding which patients should get a placebo versus active treatment. The utility of the observed data can serve as a rough esitmate for this measurement. Policies that give estimated utilities higher than this baseline will be considered superior to the historical policy.

[31]:
```
def r(action, outcome):
    '''
    Return an array with rewards of length action,
    given action and outcome are same length.
    '''
    return -0.1*np.array(action) + np.array(outcome)
```

[32]:
```
np.sum(r(atr, ytr))
```

[32]: 877.3

[33]:
```
def estimate_utility(data, actions, outcome, policy=None):
    if policy:
        actions = policy(data)

    return np.sum(r(actions, outcome))
```

[34]:
```
estimate_utility(Xtr, atr, ytr)
```

[34]: 877.3

While this value is interesting, a utility variable can contain quite a lot of noise. To get an estimate of how noisy this measurement is, we can define a confidence interval around the estimated utility of these data.

As of right now, we only have a single estimated utility. To build a confidence interval we'll need a few more estiamted utilities on these same data. Let's aggregate some subsets using a bootstrapping technique. A confidence interval over the distribution of these estimates will then give us an upper and lower error boundry of our estiamted utility, as well as the confidence of the true utility being within this interval.

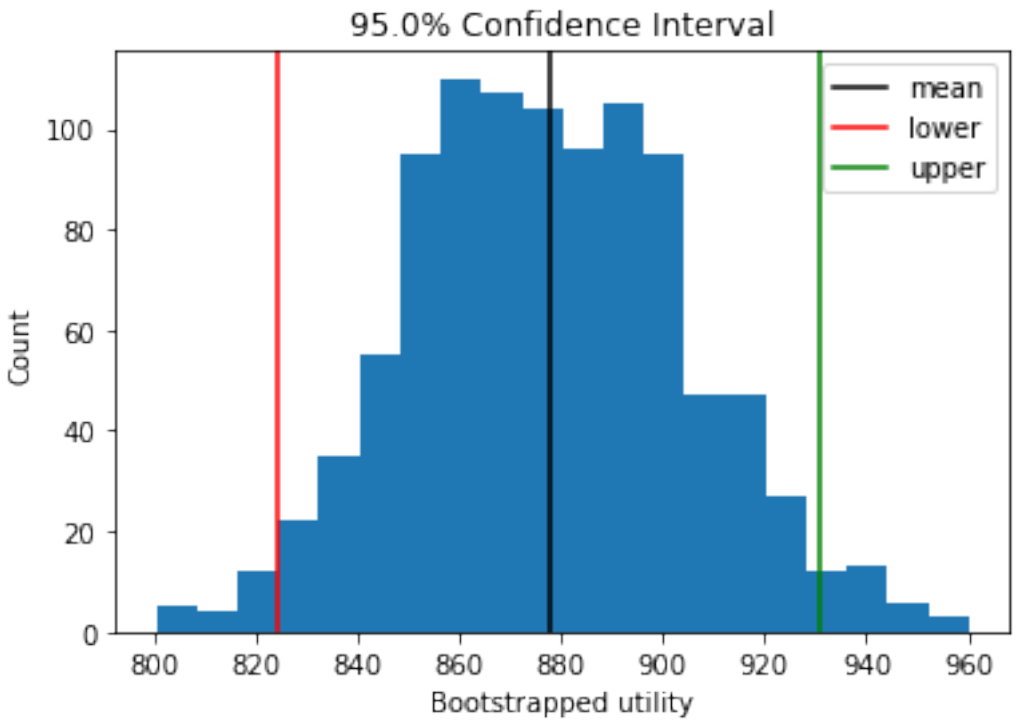
[35]:
```
B = 1000
bootstrap_util = []

for i in tqdm(range(B)):
    boot = Xtr.sample(Xtr.shape[0], replace=True)
    boot_a = atr[boot.index]
    boot_y = ytr[boot.index]
    bootstrap_util.append(estimate_utility(boot, boot_a, boot_y))
```

```
100%|                    | 1000/1000 [00:10<00:00, 99.44it/s]
```

A quick and easy way to find a confidence interval on indepently distributed varaibles can be done using the Student t continuous random variable's interval function from `scipy.stats`.

[36]:
```
import scipy.stats as st

def mean_confidence_interval(data, confidence=0.95):
    a = np.array(data)
    n = len(a)
    m, se = np.mean(a), np.std(a)
    ci = st.t.interval(alpha=confidence, df=n-1, loc=m, scale=se)
    return m, *ci
```

[37]:
```
confidence = .95
mci = mean_confidence_interval(bootstrap_util, confidence)
plt.hist(bootstrap_util, bins=20)
plt.axvline(mci[0], color='black', label='mean')
plt.axvline(mci[1], color='r', label='lower')
plt.axvline(mci[2], color='green', label='upper')
plt.xlabel('Bootstrapped utility')
plt.ylabel('Count')
plt.title(f'{confidence*100}% Confidence Interval')
plt.legend()
plt.show()
mci
```



[37]: (877.5303, 824.07160583933, 930.98899416067)

So, this tells us that we are 95% sure that our current policy π_0 gives an estiamted utility somewhere between $(850, 950)$ for our current training data. As mentioned before, a policy giving a higher estimated range of a confidence interval than this is considered better than this π_0.

4.2 Improved policies

A simple improvement we can make to our policy is to always choose the action that maximizes the utility. This means, for every data point, measure the utility of each action, then choose the action that gives the highest value for utility.

To start, we'll have to build a model for predicting the outcomes from the actions and data. Here' we'll use a random forest classifier.

We'll also need to define our prior beleif for good actions. This is done by building a classifier using the user data as our features, and the actions from policy π_0 as our target.

[50]:
```
from sklearn.ensemble import RandomForestClassifier
from sklearn.linear_model import LogisticRegression

model = RandomForestClassifier().fit(Xtr.join(atr), ytr)
prior = LogisticRegression().fit(Xtr, atr)
```

We can now define the function `improved_policy()` that will find the expected utility for both $a = 0$ and $a = 1$, then return the action giving the highest utility, for each data point. We can

write this function to accept x as an array for vectorized computations. This really speeds up the computation time, and avoids bottlenecking when working on large data sets, like ours.

[39]:
```
def improved_policy(x):   # policy of action a given data x
    pi = prior.predict_proba(x)
    pi_0 = pi[:,0]
    pi_1 = pi[:,1]
    treatments = np.ones(pi.shape[0])
    treatments[pi_0 >= pi_1] = 0    # predicted best actions based off hist. policy

    data_treat = x.copy()
    data_treat['a'] = treatments.tolist()
    P_y = model.predict_proba(data_treat)
    p_y_0 = P_y[:,0]
    p_y_1 = P_y[:,1]                # probs of getting y=1 given data and predicted a

    exp_util_0 = pi_0*(p_y_0*r(0, 0) + p_y_1*r(0, 1))
    exp_util_1 = pi_1*(p_y_0*r(0, 0) + p_y_1*r(0, 1))

    actions = np.ones_like(treatments)
    actions[exp_util_0>=exp_util_1] = 0
    return actions
```

We're now ready to get an estimate of our improved utility using this new policy $\hat{\pi}$ in place of π_0.

[40]:
```
estimate_utility(Xtr, improved_policy(Xtr), ytr)
```

[40]:
```
978.3999999999999
```

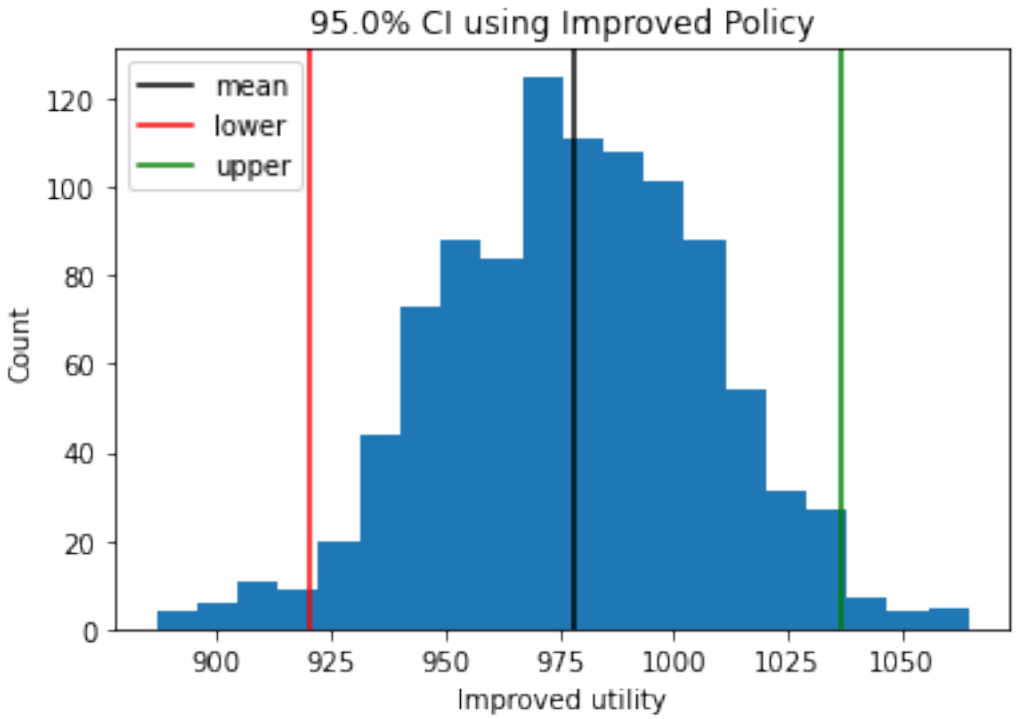
Again, this is only one estimate of utility on our data. So even though it seems much larger than the historical data, and well outside of the previous confidence interval found, we'll need to get rid of the possible noise. This is done by again finding the 95% confidence interval around this estimated utility. Comparing this interval for the interval of the historical data will tell us how much our policy actually improved.

We will use the same bootstrapping technique from earlier.

[41]:
```
B = 1000
improved_util = []

for i in tqdm(range(B)):
    boot = Xtr.sample(Xtr.shape[0], replace=True)
    boot_y = ytr[boot.index]
    improved_util.append(estimate_utility(Xtr, improved_policy(boot),
                                          boot_y))
```

```
100%|                    | 1000/1000 [02:35<00:00,  6.43it/s]
```

[42]:
```
confidence = .95
improved_mci = mean_confidence_interval(improved_util, confidence)
plt.hist(improved_util, bins=20)
plt.axvline(improved_mci[0], color='black', label='mean')
plt.axvline(improved_mci[1], color='r', label='lower')
plt.axvline(improved_mci[2], color='green', label='upper')
plt.xlabel('Improved utility')
plt.ylabel('Count')
plt.title(f'{confidence*100}% CI using Improved Policy')
plt.legend()
plt.show()
improved_mci
```



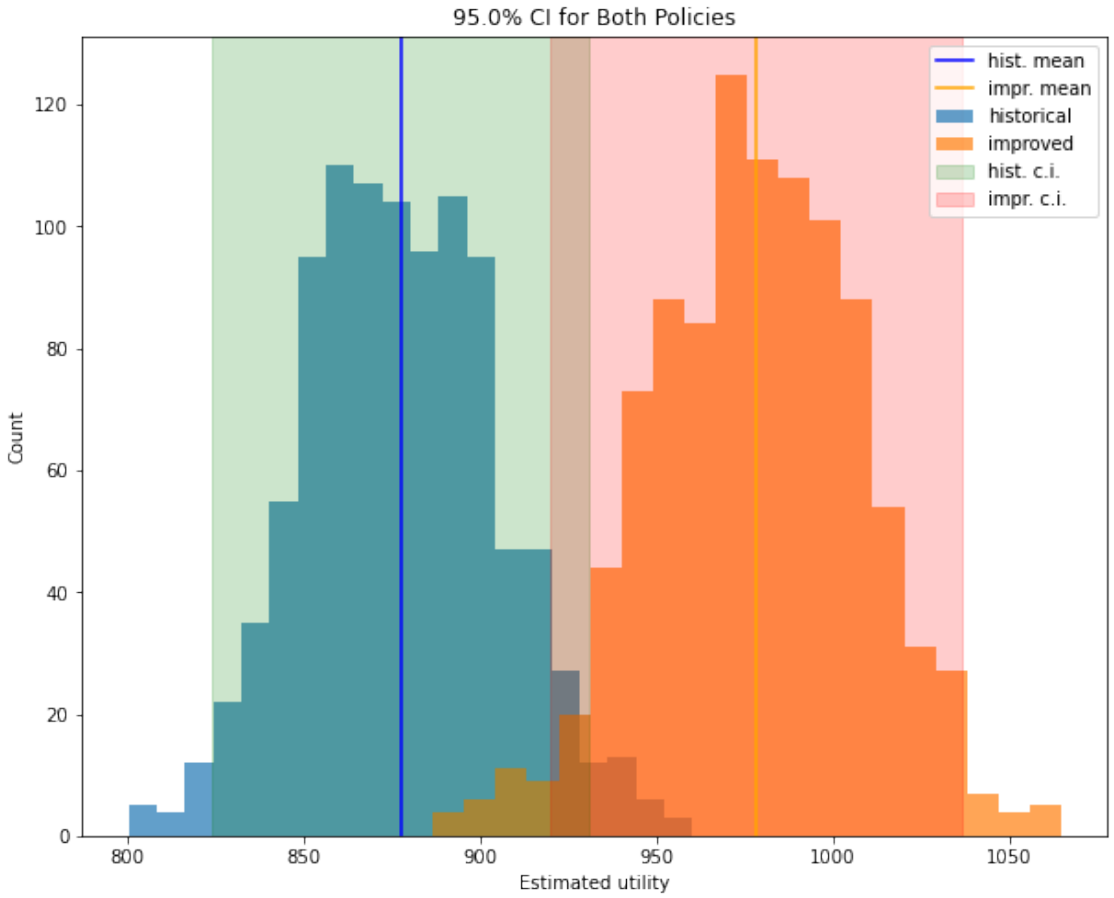
[42]:
```
(978.4268000000001, 919.980175978485, 1036.8734240215151)
```

From this 95% confidence interval, with average 980 and lower and upper boundries 920 and 1030 respectively, we can conclude that our policy gives a measurable improvement for utility than the historical policy.

We plot the two distributions with their respective confidence intervals in the same plot to highlight the amount of improvement we got.

[43]:
```
# plot both bootstrapped utility distributions on top of each other
plt.figure(figsize = (10, 8))
```

```
# dist.
plt.hist(bootstrap_util, bins=20, alpha=0.7, label='historical')
plt.hist(improved_util, bins=20,  alpha=0.7, label='improved')

# confidence intervals
plt.axvline(mci[0], color='blue', label='hist. mean')
plt.axvspan(mci[1], mci[2], color='green', alpha=0.2, label='hist. c.i.')

plt.axvline(improved_mci[0], color='orange', label='impr. mean')
plt.axvspan(improved_mci[1], improved_mci[2], color='red', alpha=0.2,␣
 ↪label='impr. c.i.')

# pynt
plt.xlabel('Estimated utility')
plt.ylabel('Count')
plt.title(f'{confidence*100}% CI for Both Policies')
plt.legend()
plt.show()

pd.DataFrame({'historical': mci, 'improved':improved_mci}, index=['mean',␣
 ↪'lower', 'upper'])
```



[43]:

	historical	improved
mean	877.530300	978.426800
lower	824.071606	919.980176
upper	930.988994	1036.873424

Remenber, these bootstrapped utilities are generated on the *exact same data.* So, the higher values of the confidence interval of the improved policy show an *actual improvment* in policy.

4.3 Unexplored alternative

Another approach for estimating the error bounds on our estimated utilities would have been to look at the Hoeffding's inequality for the data. For a two-sided interval, this is defined as:

$$P(|\mu_n - E\mu_n| \geq \epsilon) \leq 2e^{-2n\epsilon^2}$$

where μ_n is the emperical mean over our data

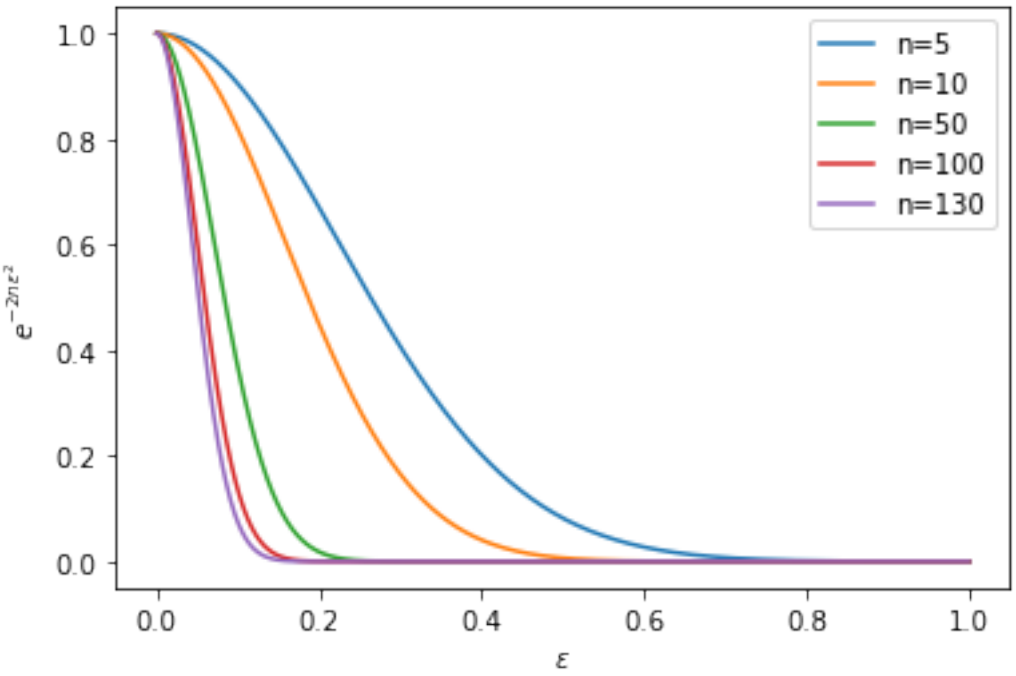
$$\mu_n \triangleq \frac{1}{n} sum_{t=1}^{n} x_t$$

and the x_is are the independently and individually distributed random variables of our data set, i.e. $i \in [1, 130]$.

Here, epsilon is the window of how much error we can allow and still have a trustworthy estimate. As the number of features of the data set increases, this window size decreases. This is visualized in the plot below.

[44]:
```
eps = np.linspace(0, 1, 1000)
for n in [5, 10, 50, 100, 130]:
    plt.plot(eps, np.exp(-2*n*eps**2), label = f'n={n}')
plt.xlabel('$\epsilon$')
plt.ylabel('$e^{-2n\epsilon ^2}$')
plt.legend()
```

[44]:
```
<matplotlib.legend.Legend at 0x20f0db34460>
```



Rearranging this inequality gives a probability $1 - \delta$ that the amount of acceptable error will be less than or equal to some constant corresponding to δ:

$$|mu_n - E\mu_n| = \sqrt{\frac{ln(2/\delta)}{2n}}$$

The inequality can give an okay estimate of the error boundries of less-complex data sets, but does not scale well for more complex models. This is why we instead choose t-static condfidence intervals to estimate these boundries. It was only added here as additional information.

5 Adaptive experiment design

We can now implement the above code into different recommender classes, to formally test the scale of improvement.

5.1 Online policy testing

The classes `HistoricalRecommender` and `ImprovedRecommender` will follow the same respective procedures for utility estimatation as shown above. See the attached files for more details.

[86]:
```
import reference_recommender as rr

historical_recommender = rr.HistoricalRecommender(n_actions=2,
                                                  n_outcomes=2)
historical_recommender.set_reward(r)
historical_recommender.fit_treatment_outcome(Xtr, atr, ytr)
historical_recommender.fit_data(Xtr.join(atr))
```

```
Preprocessing data
```

[87]:
```
historical_recommender.estimate_utility(Xtr, atr, ytr)
```

[87]:
```
877.3
```

[91]:
```
import improved_recommender as ir

improved_recommender = ir.ImprovedRecommender(n_actions=2, n_outcomes=2)
improved_recommender.set_reward(r)
improved_recommender.fit_data(Xtr.join(atr))  # ok to use classifier here?
improved_recommender.fit_treatment_outcome(Xtr, atr, ytr)
```

[67]:
```
improved_recommender.estimate_utility(Xtr, atr, ytr,
                                       improved_recommender.improved_policy)
```

[67]:
```
978.8
```

These are the same values as before, telling us methods and procedures in both `HistoricalRecommender` and `ImprovedRecommender` match what is done above.

We are now ready to run the test file for the historical and improved recommenders. (Here, the test-file was slightly altered to enable testing for the recommenders created in this walk through. The argument `improved` calls this specific test.)

[68]:
```
!python TestRecommender.py improved
```

```
Setting up simulator
---- Testing with only two treatments ----
--- Historical ---
Setting up policy
Fitting historical data to the policy
Running an online test
```

```
Total reward: -71.40000000000036
--- Improved ---
Running an online test
Total reward: 122.80000000000061
```

For historical policy on the generated data, we saw a total reward in the range of $(-75, -50)$.

The improved policy gave a positive total reward, with a range of $(125, 150)$.

This again shows that our implementation for a policy chosing the action that maximizes the utility is much better than the policy used to generate this data set.

5.2 Adaptive experiments

We'd now like to make a model based on reinforcement learning, which updates it's recommendations as it observes new data.

This creates two different goals we can work to solve: (a) discover the most effective policy at the end of the trail or (b) maximize the number of people who recover.

For simplicity, we will stick to our original goal of maximizing the total number of people who recoverd.

To implement this recommender, we needed to add a method `observe()` to our recommender class. This method takes in a single data point, gets teh recommended action for this data point, then compares this action to the true action from the data set.

If the true action gives a higher reward than the estimated action, our model is the refit to incorperate this new information. To avoid unnecessary computations, refitting is skipped for the recommendations that match the true values for the actions.

The results of this implementation are tested below.

[83]:
```
!python TestRecommender.py improved adaptive
```

```
Setting up simulator
---- Testing with only two treatments ----
--- Historical ---
Setting up policy
Fitting historical data to the policy
Running an online test
Total reward: -81.59999999999928
--- Improved ---
Running an online test
Total reward: 120.40000000000052
--- Adaptive ---
Setting up policy
Fitting historical data to the policy
Running an online test
Total reward: 149.00000000000068
```

We see a slight increase in total reward here, although not nearly as much as when we improved our policy. And just like before, since we know utility is a noisy varaible, this single estimate doesn't

provide enough confidence of actual improvement.

Let's try to generate the data mutliple times, and see if we can build confidence intervals around these results.

To do this, we'll rewrite the needed code from `TestRecommender.py` here to see easier follow along and see what is going on.

[70]:
```
from TestRecommender import default_reward_function, test_policy
import data_generation
import reference_recommender as rr
import improved_recommender as ir
import adaptive_recommender as ar


B = 50
results = []
for b in tqdm(range(B)):
    generator = data_generation.DataGenerator(matrices="./
 ↪big_generating_matrices.mat")

    historic_policy = rr.HistoricalRecommender
    improved_policy = ir.ImprovedRecommender
    adaptive_policy = ar.AdaptiveRecommender

    iter_results = []
    for policy_factory in [historic_policy, improved_policy, adaptive_policy]:
        policy = policy_factory(generator.get_n_actions(), generator.
 ↪get_n_outcomes())
        policy.fit_treatment_outcome(Xtr, atr, ytr)
        n_tests = 100 # a magnitude smaller for faster calculations
        result = test_policy(generator, policy, default_reward_function,␣
 ↪n_tests)
        iter_results.append(result)
    results.append(iter_results)
```

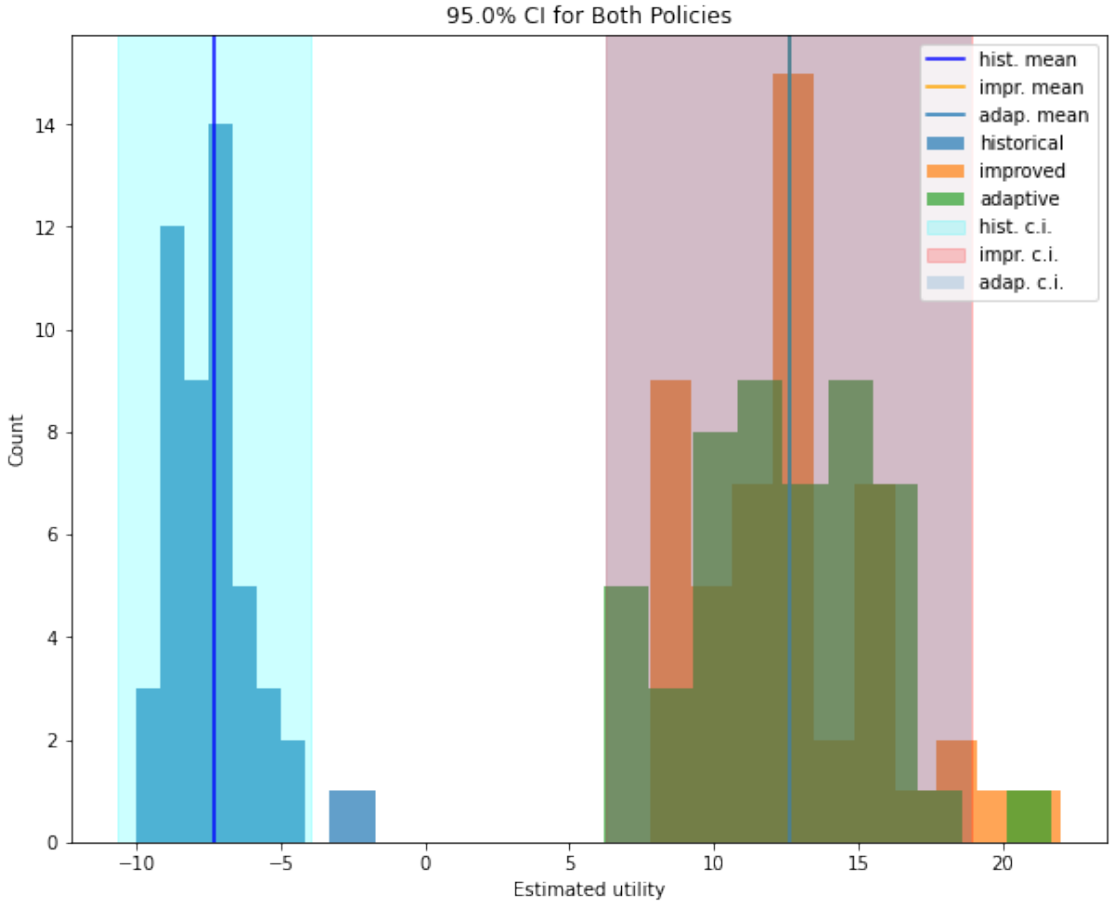
```
100%|                    | 50/50 [05:15<00:00,  6.31s/it]
```

[71]:
```
hist_results = np.array(results)[:,0]
impr_results = np.array(results)[:,1]
adap_results = np.array(results)[:,2]
```

[72]:
```
hist_mci = mean_confidence_interval(hist_results)
impr_mci = mean_confidence_interval(impr_results)
adap_mci = mean_confidence_interval(adap_results)
```

[73]:
```
# plot both bootstrapped utility distributions on top of each other
plt.figure(figsize = (10, 8))
```

```
# dist.
plt.hist(hist_results, alpha=0.7, label='historical')
plt.hist(impr_results,  alpha=0.7, label='improved')
plt.hist(adap_results,  alpha=0.7, label='adaptive')

# confidence intervals
plt.axvline(hist_mci[0], color='blue', label='hist. mean')
plt.axvspan(hist_mci[1], hist_mci[2], color='cyan', alpha=0.2, label='hist. c.i.
 ↪')

plt.axvline(impr_mci[0], color='orange', label='impr. mean')
plt.axvspan(impr_mci[1], impr_mci[2], color='red', alpha=0.2, label='impr. c.i.
 ↪')

plt.axvline(impr_mci[0], label='adap. mean')
plt.axvspan(impr_mci[1], impr_mci[2], alpha=0.2, label='adap. c.i.')


# pynt
plt.xlabel('Estimated utility')
plt.ylabel('Count')
plt.title(f'{confidence*100}% CI for Both Policies')
plt.legend()
plt.show()

# table with values
pd.DataFrame({'historical': hist_mci,
              'improved': impr_mci,
              'adaptive': adap_mci},
             index=['mean', 'lower', 'upper'])
```



[73]:

	historical	improved	adaptive
mean	-7.324000	12.606000	12.546000
lower	-10.646202	6.267657	6.050723
upper	-4.001798	18.944343	19.041277

From the looks of the above confidence intervals, we can conclude (again) that we gain little to no efficiency when using our implementation of an adaptive model. Since the adaptive recommender take much longer to compute, and is much more complex, our `ImprovedRecommender` would be the desired decision maker for these data.

6 Summary

In this project, we: (1) looked at the causality between data, actions, and final outcomes, (2) measured the utility of the historical actions versus those derived from an improved policy, and (3) built both fixed and adaptive recommendation systems in order to maximize this utility.

In conclusion, the best recommendation system had a policy of recommending the actions that would maximize the expected utility, from a fixed model. The adaptive model was disregarded due to high computational cost, will miniscule efficiecy gains.

The most interesting genes were found using a forward selection model. These included genes $4, 1, 9, 36, 125, 6, 79$, and 87. The patients gender and smoking habits also were considered important for estiamting both symptoms.

The advantage of gene-targeting treatments over some arbitrary fixed treatment (always $a = 0$ or always $a = 1$) arises when a utility function (reward) is implemented. This function penalizes active treatment and rewards recoveries (with seperate respective weights). The utility function reflects real-world situations, where unknown side-effects of active treatments "penalize" the users. To show this on our data, we could estiamte the utility of only choosing a single a for all of the data points, and comparing to the estimated utilities found above. (Done in `final_analysis`.)

A final analysis of the recommender classes can be observed by calling `recommender.final_analysis()`.

[95]:
```
improved_recommender.final_analysis()
```

```
--- Final Analysis ---
 Features important for symptoms (with cross validated accuracy means)
gene4: (0.70027)
gender: (0.70027)
smoker: (0.70027)
gene1: (0.70027)
gene93: (0.7012)
gene121: (0.70227)
gene66: (0.70227)
gene31: (0.7008)
gene64: (0.69907)
gene24: (0.69293)

 Estimated historic utility
877.3

 Estimated improved utility
978.3

 Compare to fixed treatment a=0
1049.0

 Compare to fixed treatment a=1
299.0


Total recoveries: 1049
 out of 7500 patients
```

From the final analysis above, we see that the policy giving the highest total utility in the end is the fixed treatment of only placebo $a = 0$. However, this just tells us the number of recovered patients in the 7500 data points included given to the recommender. Our improved recommender gave a relatively high reward, by treating only those patients who were likely to actually recover.

Further analysis could explore the features found in the feature selection, but also work on defining an even better policy, or testing out different utility functions.

[ ]: