

✓ Support Vector Machines in Python

Task 1: Import the modules that will do all the work

```
import pandas as pd # pandas is used to load and manipulate data and for One-Hot Enc
import numpy as np # numpy is used to calculate the mean and standard deviation
import matplotlib.pyplot as plt # matplotlib is for drawing graphs
import matplotlib.colors as colors
from sklearn.model_selection import train_test_split # split data into training and
from sklearn.preprocessing import scale # scale and center data
from sklearn.svm import SVC # this will make a support vector machine for classifica
from sklearn.model_selection import GridSearchCV # this will do cross validation
from sklearn.metrics import confusion_matrix # this creates a confusion matrix
# Instead of importing plot_confusion_matrix, import ConfusionMatrixDisplay
from sklearn.metrics import ConfusionMatrixDisplay # draws a confusion matrix
from sklearn.decomposition import PCA # to perform PCA to plot the data
```

✓ Task 2: Import the data - UCI Heart disease Dataset!

```
df = pd.read_csv("processed.cleveland.data", header=None)
```

```
df.head()
```



	0	1	2	3	4	5	6	7	8	9	10	11	12	13
0	63.0	1.0	1.0	145.0	233.0	1.0	2.0	150.0	0.0	2.3	3.0	0.0	6.0	0
1	67.0	1.0	4.0	160.0	286.0	0.0	2.0	108.0	1.0	1.5	2.0	3.0	3.0	2
2	67.0	1.0	4.0	120.0	229.0	0.0	2.0	129.0	1.0	2.6	2.0	2.0	7.0	1
3	37.0	1.0	3.0	130.0	250.0	0.0	0.0	187.0	0.0	3.5	3.0	0.0	3.0	0
4	41.0	0.0	2.0	130.0	204.0	0.0	2.0	172.0	0.0	1.4	1.0	0.0	3.0	0

We see that instead of nice column names, we just have column numbers. Since nice column names would make it easier to know how to format the data, let's replace the column numbers with the following column names:

age,

sex,

cp - chest pain,

restbp - resting blood pressure (in mm Hg),

chol - serum cholesterol in mg/dl,

fbs - fasting blood sugar,

restecg - resting electrocardiographic results,

thalach - maximum heart rate achieved,

exang - exercise induced angina,

oldpeak - ST depression induced by exercise relative to rest,

slope - the slope of the peak exercise ST segment,

ca - number of major vessels (0-3) colored by fluoroscopy,

thal - this is short of thalium heart scan,

hd - diagnosis of heart disease, the predicted attribute.

```
df.columns = ['age',
              'sex',
              'cp',
              'restbp',
              'chol',
              'fbs',
              'restecg',
              'thalach',
              'exang',
              'oldpeak',
              'slope',
              'ca',
              'thal',
              'hd']
```

```
df.head()
```



	age	sex	cp	restbp	chol	fbs	restecg	thalach	exang	oldpeak	slope	ca	thal
0	63.0	1.0	1.0	145.0	233.0	1.0	2.0	150.0	0.0	2.3	3.0	0.0	
1	67.0	1.0	4.0	160.0	286.0	0.0	2.0	108.0	1.0	1.5	2.0	3.0	
2	67.0	1.0	4.0	120.0	229.0	0.0	2.0	129.0	1.0	2.6	2.0	2.0	
3	37.0	1.0	3.0	130.0	250.0	0.0	0.0	187.0	0.0	3.5	3.0	0.0	

We have replaced the column numbers with nice, easy to remember names. Now that we have the data in a data frame called `df`, we are ready to identify and deal with Missing Data.

✓ Task 3: Missing Data Part 1: Identifying Missing Data

Missing Data is simply a blank space or surrogate value that indicates that we failed to collect data for one of the features. For example, if we forgot to ask someone's age, or forgot to write it down, then we would have a blank space in the dataset for that person's age.

There are two main ways to deal with missing data:

We can remove the rows that contain missing data from the dataset. This is relatively easy to do, but it wastes all of the other values that we collected. How big of a waste this is depends on how important this missing value is for classification. For example, if we are missing a value for age, and age is not useful for classifying if people have heart disease or not, then it would be a shame to throw out all of someone's data just because we do not have their age. We can impute the values that are missing. In this context impute is just a fancy way of saying "we can make an educated guess about about what the value should be". Continuing our example where we are missing a value for age, instead of throwing out the entire row of data, we can fill the missing value with the average age or the median age, or use some other, more sophisticated approach, to guess at an appropriate value. In this section, we'll focus on identifying missing values in the dataset and dealing with them.

First, let's see what sort of data is in each column.

```
df.dtypes
```



0

age	float64
sex	float64
cp	float64
restbp	float64
chol	float64
fbs	float64
restecg	float64
thalach	float64
exang	float64
oldpeak	float64
slope	float64
ca	object
thal	object
hd	int64

dtype: object

We see that that they are almost all float64, however, two columns, ca and thal, have the object type and one column, hd has int64.

The fact that the ca and thal columns have object data types suggests there is something funny going on in them. object datatypes are used when there are mixtures of things, like a mixture of numbers and letters. In theory, both ca and thal should just have a few values representing different categories, so let's investigate what's going on by printing out their unique values. We'll start with ca:

```
df['ca'].unique()
```



```
array(['0.0', '3.0', '2.0', '1.0', '?'], dtype=object)
```

We see that ca contains numbers (0.0, 3.0, 2.0 and 1.0) and questions marks (?). The numbers represent the number of blood vessels that we lit up by fluoroscopy and the question marks represent missing data.

Now let's look at the unique values in thal.

```
df['thal'].unique()
```

```
array(['6.0', '3.0', '7.0', '?'], dtype=object)
```

Again, thal also contains a mixture of numbers, representing the different diagnoses from the thalium heart scan, and question marks, which represent missing values.

✓ Task 4: Missing Data Part 2: Dealing With Missing Data

Since scikit-learn's support vector machines do not support datasets with missing values, we need to figure out what to do these question marks. We can either delete these patients from the training dataset, or impute values for the missing data. First let's see how many rows contain missing values.

```
len(df.loc[(df['ca']=='?')|(df['thal']=='?')])
```

```
6
```

Since only 6 rows have missing values, let's look at them.

```
df.loc[(df['ca'] == '?') | (df['thal'] == '?')]
```

	age	sex	cp	restbp	chol	fbs	restecg	thalach	exang	oldpeak	slope	ca
87	53.0	0.0	3.0	128.0	216.0	0.0	2.0	115.0	0.0	0.0	1.0	0.0
166	52.0	1.0	3.0	138.0	223.0	0.0	0.0	169.0	0.0	0.0	1.0	?
192	43.0	1.0	4.0	132.0	247.0	1.0	2.0	143.0	1.0	0.1	2.0	?
266	52.0	1.0	4.0	128.0	204.0	1.0	0.0	156.0	1.0	1.0	2.0	0.0
287	58.0	1.0	2.0	125.0	220.0	0.0	0.0	144.0	0.0	0.4	2.0	?

Now let's count the number of rows in the full dataset.

```
len(df)
```

```
303
```

So 6 of the 303 rows, or 2%, contain missing values. Since $303 - 6 = 297$, and 297 is plenty of data to build a support vector machine, we will remove the rows with missing values, rather than try to impute their values. We do this by selecting all of the rows that do not contain question marks in either the ca or thal columns:

```
df_no_missing=df.loc[(df['ca']!='?') & (df['thal']!='?')]
```

Since df_no_missing has 6 fewer rows than the original df, it should have 297 rows.

```
len(df_no_missing)
```

```
297
```

```
df_no_missing['ca'].unique()
```

```
array(['0.0', '3.0', '2.0', '1.0'], dtype=object)
```

And we can also do the same thing for thal:

```
df_no_missing['thal'].unique()
```

```
array(['6.0', '3.0', '7.0'], dtype=object)
```

We have verified that df_no_missing does not contain any missing values. NOTE: ca and thal still have the object data type. That's OK. Now we are ready to format the data for making a Support Vector Machine.

✓ Task 5: Format Data Part 1: Split the Data into Dependent and Independent Variables

Now that we have taken care of the missing data, we are ready to start formatting the data for making a Support Vector Machine.


The first step is to split the data into two parts:

The columns of data that we will use to make classifications The column of data that we want to predict. We will use the conventional notation of X (capital X) to represent the columns of data that

we will use to make classifications and y (lower case y) to represent the thing we want to predict. In this case, we want to predict hd (heart disease).

NOTE: In the code below we are using `copy()` to copy the data by value. By default, pandas uses copy by reference. Using `copy()` ensures that the original data `df_no_missing` is not modified when we modify X or y . In other words, if we make a mistake when we are formatting the columns for support vector machines, we can just re-copy `df_no_missing`, rather than have to reload the original data and remove the missing values etc.

```
X=df_no_missing.drop('hd',axis=1).copy()
X.head()
```



	age	sex	cp	restbp	chol	fbs	restecg	thalach	exang	oldpeak	slope	ca	th
0	63.0	1.0	1.0	145.0	233.0	1.0	2.0	150.0	0.0	2.3	3.0	0.0	
1	67.0	1.0	4.0	160.0	286.0	0.0	2.0	108.0	1.0	1.5	2.0	3.0	
2	67.0	1.0	4.0	120.0	229.0	0.0	2.0	129.0	1.0	2.6	2.0	2.0	
3	37.0	1.0	3.0	130.0	250.0	0.0	0.0	187.0	0.0	3.5	3.0	0.0	

```
y = df_no_missing['hd'].copy()
y.head()
```



	hd
0	0
1	2
2	1
3	0
4	0

dtype: int64

✓ Task 6: Format the Data Part 2: One-Hot Encoding

Now that we have split the data frame into two pieces, X , which contains the data we will use to make, or predict, classifications, and y , which contains the known classifications in our training dataset, we need to take a closer look at the variables in X . The list bellow tells us what each variable represents and the type of data (float or categorical) it should contain:

age, Float

sex - Category

0 = female 1 = male

cp, chest pain, Category

1 = typical angina, 2 = atypical angina, 3 = non-anginal pain, 4 = asymptomatic

restbp, resting blood pressure (in mm Hg), Float

chol, serum cholesterol in mg/dl, Float

fbs, fasting blood sugar, Category

0 = ≥ 120 mg/dl 1 = < 120 mg/dl

restecg, resting electrocardiographic results, Category

1 = normal 2 = having ST-T wave abnormality 3 = showing probable or definite left ventricular hypertrophy

thalach, maximum heart rate achieved, Float

exang, exercise induced angina, Category

0 = no 1 = yes

oldpeak, ST depression induced by exercise relative to rest, Float

slope, the slope of the peak exercise ST segment, Category

1 = upsloping 2 = flat 3 = downsloping

ca, number of major vessels (0-3) colored by fluoroscopy, Float

thal, thalium heart scan, Category

3 = normal (no cold spots) 6 = fixed defect (cold spots during rest and exercise) 7 = reversible defect (when cold spots only appear during exercise)

Now, just to review, let's look at the data types in X to remember how python is seeing the data right now.

X.dtypes



0

age	float64
sex	float64
cp	float64
restbp	float64
chol	float64
fbs	float64
restecg	float64
thalach	float64
exang	float64
oldpeak	float64
slope	float64
ca	object
thal	object

dtype: object`X['cp'].unique()``array([1., 4., 3., 2.])`

we will convert it, using One-Hot Encoding, into a series of columns that only contains 0s and 1s.

If we treated these values, 1, 2, 3 and 4, like continuous data, then we would assume that 4, which means "asymptomatic", is more similar to 3, which means "non-anginal pain", than it is to 1 or 2, which are other types of chest pain. Thus, the support vector machine would be more likely to cluster the patients with 4s and 3s together than the patients with 4s and 1s together. In contrast, if we treat these numbers like categorical data, then we treat each one a separate category that is no more or less similar to any of the other categories. Thus, the likelihood of clustering patients with 4s with 3s is the same as clustering 4s with 1s, and that approach is more reasonable.

`pd.get_dummies(X, columns=['cp']).head()`



	age	sex	restbp	chol	fbs	restecg	thalach	exang	oldpeak	slope	ca	thal
0	63.0	1.0	145.0	233.0	1.0	2.0	150.0	0.0	2.3	3.0	0.0	6.0
1	67.0	1.0	160.0	286.0	0.0	2.0	108.0	1.0	1.5	2.0	3.0	3.0
2	67.0	1.0	120.0	229.0	0.0	2.0	129.0	1.0	2.6	2.0	2.0	7.0
3	37.0	1.0	130.0	250.0	0.0	0.0	187.0	0.0	3.5	3.0	0.0	3.0
4	41.0	0.0	130.0	204.0	0.0	0.0	172.0	0.0	1.4	3.0	0.0	3.0

```
X_encoded=pd.get_dummies(X, columns=['cp','restecg','slope','thal'])
X_encoded.head()
```



	age	sex	restbp	chol	fbs	thalach	exang	oldpeak	ca	cp_1.0	...	cp_4.0	r
0	63.0	1.0	145.0	233.0	1.0	150.0	0.0	2.3	0.0	True	...	False	
1	67.0	1.0	160.0	286.0	0.0	108.0	1.0	1.5	3.0	False	...	True	
2	67.0	1.0	120.0	229.0	0.0	129.0	1.0	2.6	2.0	False	...	True	
3	37.0	1.0	130.0	250.0	0.0	187.0	0.0	3.5	0.0	False	...	False	
4	41.0	0.0	130.0	204.0	0.0	172.0	0.0	1.4	0.0	False	...	False	

y doesn't just contain 0s and 1s. Instead, it has 5 different levels of heart disease. 0 = no heart disease and 1-4 are various degrees of heart disease. We can see this with `unique()`:

```
y.unique()
```



```
array([0, 2, 1, 3, 4])
```

Since we're only making a support vector machine that does simple classification and only care if someone has heart disease or not, we need to convert all numbers > 0 to 1.

```
y_not_zero_idx=y>0
y[y_not_zero_idx]=1
y.unique()
```



```
array([0, 1])
```

✓ Task 7: Format the Data Part 3: Centering and Scaling

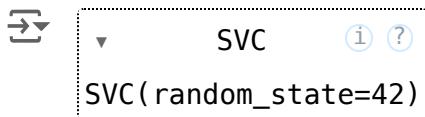
The Radial Basis Function (RBF) that we are using with our Support Vector Machine assumes that the data are centered and scaled, so we need to do this to both the training and testing datasets.

NOTE: We split the data into training and testing datasets and then scale them separately to avoid Data Leakage. Data Leakage occurs when information about the training dataset corrupts or influences the testing dataset.

```
X_train, X_test, y_train, y_test=train_test_split(X, y, random_state=42)
X_train_scaled=scale(X_train)
X_test_scaled=scale(X_test)
```

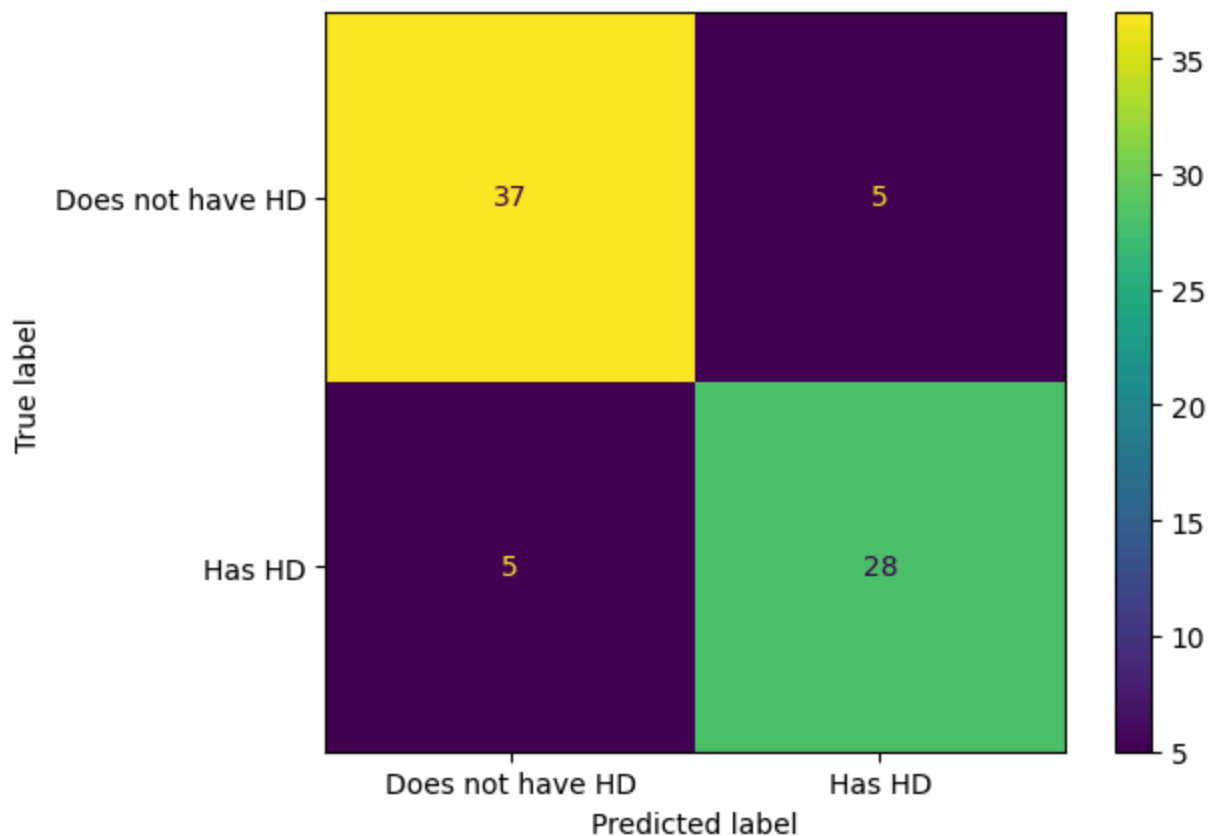
✓ Task 8: Build A Preliminary Support Vector Machine

```
clf_svm=SVC(random_state=42)
clf_svm.fit(X_train_scaled, y_train)
```



we've built a Support Vector Machine for classification. Let's see how it performs on the Testing Dataset and draw a Confusion Matrix.

```
cm = ConfusionMatrixDisplay.from_estimator(
    clf_svm,
    X_test_scaled,
    y_test,
    display_labels=["Does not have HD", "Has HD"]
)
```



In the confusion matrix, we see that of the $37 + 5 = 42$ people that did not have Heart Disease, 37 (88%) were correctly classified. And of the $5 + 28 = 33$ people that have Heart Disease, 28 (85%) were correctly classified. So the support vector machine did pretty well without any optimization. That said, it is possible that we can improve predictions using Cross Validation to optimize the parameters.

✓ Task 9: Optimize Parameters with Cross Validation

Optimizing a Support Vector Machine is all about finding the best value for gamma, and, potentially, the regularization parameter, C. So let's see if we can find better parameters using cross validation in hopes that we can improve the accuracy with the Testing Dataset.

Since we have to parameters two optimize, we will use `GridSearchCV()`. We do this by specifying potential values for gamma and C, and `GridSearchCV()` tests all possible combinations of the parameters for us.

```
param_grid=[
    {'C': [1,10,100,1000],
     'gamma': [0.001, 0.0001],
     'kernel': ['rbf']},
```

```

]
optimal_params = GridSearchCV(
    SVC(),
    param_grid,
    cv=5,
    verbose=0
)
optimal_params.fit(X_train_scaled, y_train)
optimal_params.best_params_

{ 'C': 10, 'gamma': 0.001, 'kernel': 'rbf' }

```

we see that the ideal value for C is 10 and the ideal value for gamma is 0.001.

✓ Task 10: Building, Evaluating, Drawing, and Interpreting the Final Support Vector Machine

```

clf_svm=SVC(random_state=42, C=10, gamma=0.001)
clf_svm.fit(X_train_scaled, y_train)

```

```

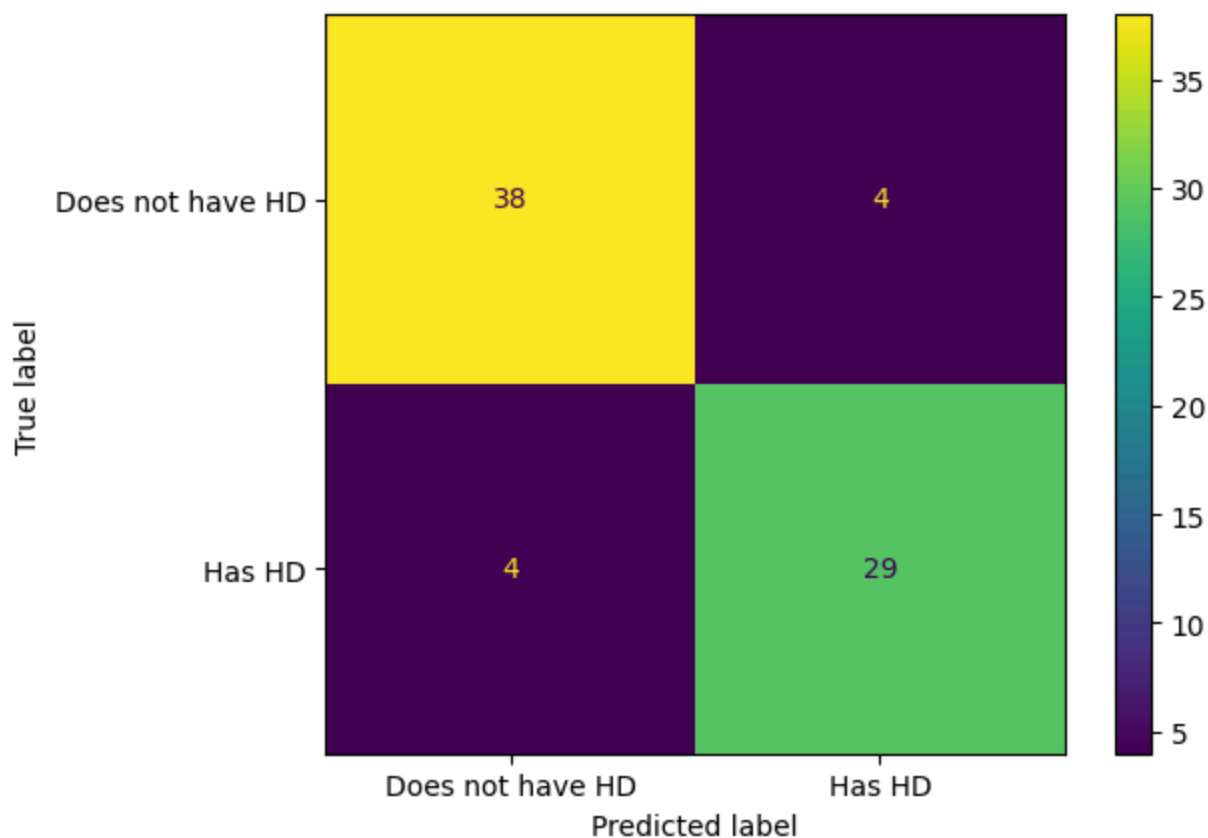
SVC
SVC(C=10, gamma=0.001, random_state=42)

```

```

cm = ConfusionMatrixDisplay.from_estimator(
    clf_svm,
    X_test_scaled,
    y_test,
    display_labels=["Does not have HD", "Has HD"]
)

```



We see that the optimized Support Vector Machine is better at classifying patients than the preliminary support vector machine.

Of the $38 + 4 = 42$ people that did not have heart disease, 38 (90%) were correctly classified. This is an improvement over the preliminary support vector machine, which only correctly classified 37 (88%) of the patients without heart disease. Of the $4 + 29 = 33$ people with heart disease, 29 (87%) were correctly classified. This is also an improvement over the preliminary support vector machine, which only correctly classified 28 (85%) of the patients with heart disease. Yay for optimizing the parameters!

The last thing we are going to do is draw the optimized support vector machine decision boundary and discuss how to interpret it.

The first thing we need to do is count the number of columns in X:

```
len(df.columns)
```



14

So we see that there are 14 features, or columns, in X. This is a problem because it would require a 14-dimensional graph, one dimension per feature used to make predictions, to plot the data in its raw form. If we wanted to, we could just pick two features at random to use as x and y-axes on our

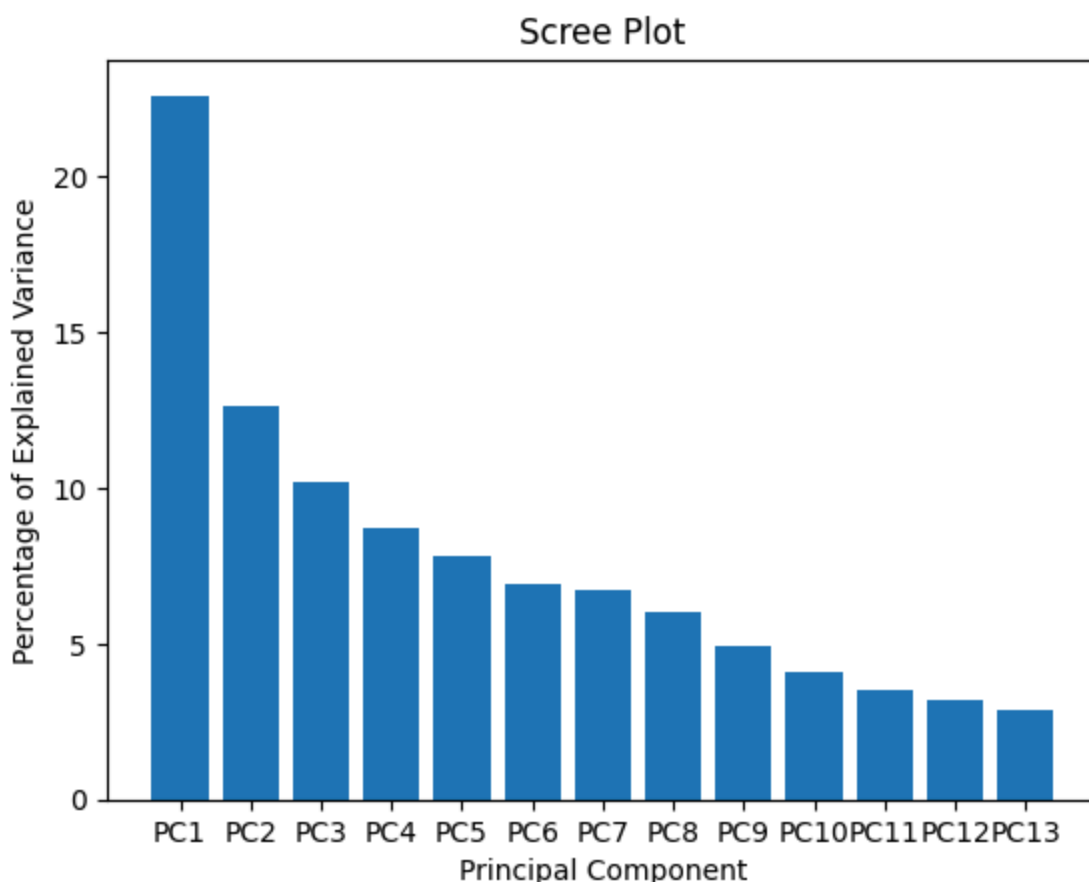
graph, but instead, we will use PCA (Principal Component Analysis) to combine the 14 features into 2 orthogonal meta-features that we can use as axes for a graph. If you don't already know about PCA, don't sweat it. For now, just know that it is a way to shrink a 14-dimensional graph into a 2-dimensional graph.

However, before we shrink the graph, let's first determine how accurate the shrunken graph will be. If it's relatively accurate, then it makes sense to draw the 2-Dimensional graph. If not, the shrunken graph will not be very useful. We can determine the accuracy of the graph by drawing something called a scree plot.

```
pca = PCA() # By default, PCA() centers the data, but does not scale it.  
X_train_pca = pca.fit_transform(X_train_scaled)
```

```
per_var = np.round(pca.explained_variance_ratio_* 100, decimals=1)  
labels = ['PC' + str(x) for x in range(1, len(per_var)+1)]
```

```
plt.bar(x=range(1, len(per_var)+1), height=per_var, tick_label=labels)  
plt.ylabel('Percentage of Explained Variance')  
plt.xlabel('Principal Component')  
plt.title('Scree Plot')  
plt.show()
```



The scree plot shows that the first principal component, PC1, accounts for a relatively large amount of variation in the raw data, and this means that it will be a good candidate for the x-axis in the 2-dimensional graph. Since PC2 accounts for the next largest amount of variance, we will use that for the y-axis.

```
pc1 = X_train_pca[:, 0]
pc2 = X_train_pca[:, 1]

## NOTE:
## pc1 contains the x-axis coordinates of the data after PCA
## pc2 contains the y-axis coordinates of the data after PCA

## Now we fit the SVM to the x and y-axis coordinates
## of the data after PCA dimension reduction...
clf_svm.fit(np.column_stack((pc1, pc2)), y_train)

## Now create a matrix of points that we can use to show
## the decision regions.
## The matrix will be a little bit larger than the
## transformed PCA points so that we can plot all of
## the PCA points on it without them being on the edge
x_min = pc1.min() - 1
x_max = pc1.max() + 1

y_min = pc2.min() - 1
y_max = pc2.max() + 1

xx, yy = np.meshgrid(np.arange(start=x_min, stop=x_max, step=0.1),
                     np.arange(start=y_min, stop=y_max, step=0.1))

## now we will classify every point in that
## matrix with the SVM. Points on one side of the
## classification boundary will get 0, and points on the other
## side will get 1.
Z = clf_svm.predict(np.column_stack((xx.ravel(), yy.ravel())))
## Right now, Z is just a long array of lots of 0s and 1s, which
## reflect how each point in the mesh was classified.
## We use reshape() so that each classification (0 or 1) corresponds
## to a specific point in the matrix.
Z = Z.reshape(xx.shape)

fig, ax = plt.subplots(figsize=(10,10))
## now we will use contourf() to draw a filled contour plot
## using the matrix values and classifications.
## The contours will be filled according to the
## predicted classifications (0s and 1s) in Z
ax.contourf(xx, yy, Z, alpha=0.1)

## now create custom colors for the actual data points
```



```
cmap = colors.ListedColormap(['#e41a1c', '#4daf4a'])
## now darw the actual data points - these will
## be colored by their known (not predcited) classifications
## NOTE: setting alpha=0.7 lets us see if we are covering up a point
scatter = ax.scatter(pc1, pc2, c=y_train,
                    cmap=cmap,
                    s=100,
                    edgecolors='k', ## 'k' = black
                    alpha=0.7)

## now create a legend
legend = ax.legend(scatter.legend_elements()[0],
                  scatter.legend_elements()[1],
                  loc="upper right")
legend.get_texts()[0].set_text("No HD")
legend.get_texts()[1].set_text("Yes HD")

## now add axis labels and titles
ax.set_ylabel('PC2')
ax.set_xlabel('PC1')
ax.set_title('Decison surface using the PCA transformed/projected features')
# plt.savefig('svm.png')
plt.show()
```



Decison surface using the PCA transformed/proiected features