Support Vector Machines

Introduction

We will be using the wine quality data set for these exercises. This data set contains various chemical properties of wine, such as acidity, sugar, pH, and alcohol. It also contains a quality metric (3-9, with highest being better) and a color (red or white). The name of the file is `Wine_Quality_Data.csv`.

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

- Import the data.
- Create the target variable `y` as a 1/0 column where 1 means red.
- Create a `pairplot` for the dataset.
- Create a bar plot showing the correlations between each column and `y`
- Pick the most 2 correlated fields (using the absolute value of correlations) and create `X`
- Use MinMaxScaler to scale `X`. Note that this will output a np.array. Make it a DataFrame again and rename the columns appropriately.

In [2]:
```
filepath = 'Wine_Quality_Data.csv'
data = pd.read_csv("C:\\Users\\rsnen\\Desktop\\IBM\\Wine_Quality_Data.csv", sep=',')
```

In [8]:
```
data.head()
```

Out[8]:

	fixed_acidity	volatile_acidity	citric_acid	residual_sugar	chlorides	free_sulfur_dioxide	total_sulfur_dioxide	density	pH	sulphates	alcohol	quality	color
0	7.4	0.70	0.00	1.9	0.076	11.0	34.0	0.9978	3.51	0.56	9.4	5	red
1	7.8	0.88	0.00	2.6	0.098	25.0	67.0	0.9968	3.20	0.68	9.8	5	red
2	7.8	0.76	0.04	2.3	0.092	15.0	54.0	0.9970	3.26	0.65	9.8	5	red
3	11.2	0.28	0.56	1.9	0.075	17.0	60.0	0.9980	3.16	0.58	9.8	6	red
4	7.4	0.70	0.00	1.9	0.076	11.0	34.0	0.9978	3.51	0.56	9.4	5	red

In [9]:
```
data.shape
```

Out[9]:
```
(6497, 13)
```

In [13]:
```
data.dtypes
```

Out[13]:
```
fixed_acidity           float64
volatile_acidity        float64
citric_acid             float64
residual_sugar          float64
chlorides               float64
free_sulfur_dioxide     float64
total_sulfur_dioxide    float64
density                 float64
pH                      float64
sulphates               float64
alcohol                 float64
quality                   int64
color                    object
dtype: object
```

In [14]:
```
#data.color equals red, and we see that that passes out true or false.
data.color == 'red'
```

Out[14]:
```
0        True
1        True
2        True
3        True
4        True
        ...
6492    False
6493    False
6494    False
6495    False
6496    False
Name: color, Length: 6497, dtype: bool
```

In [15]:
```
# When we say astype(int)
#and we see that that passes out true or false.
#we're just going to translate each one of those trues as 1 and each one of those falses as 0.

(data.color == 'red').astype(int)
```

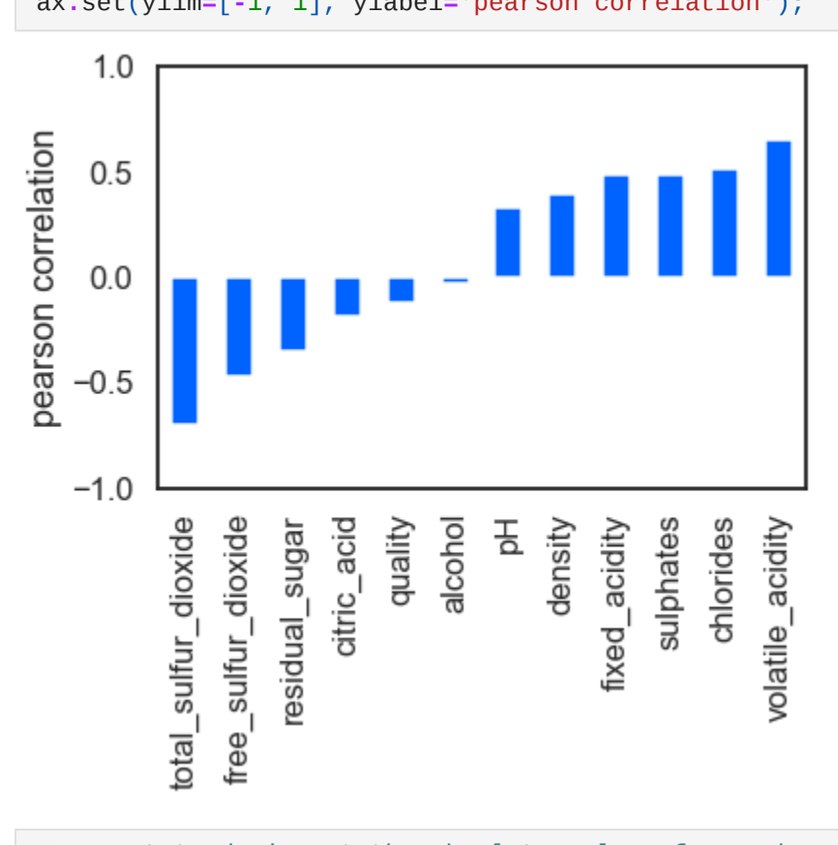
Out[15]:
```
0       1
1       1
2       1
3       1
4       1
       ..
6492    0
6493    0
6494    0
6495    0
6496    0
Name: color, Length: 6497, dtype: int32
```

In [16]:
```
y = (data['color'] == 'red').astype(int)
fields = list(data.columns[:-1])  # everything except last column i.e "color"
correlations = data[fields].corrwith(y)
correlations.sort_values(inplace=True) # sort values from smallest to largest
correlations
```

Out[16]:
```
total_sulfur_dioxide   -0.700357
free_sulfur_dioxide    -0.471644
residual_sugar         -0.348821
citric_acid            -0.187397
quality                -0.119323
alcohol                -0.032970
pH                      0.329129
density                 0.390645
fixed_acidity           0.486740
sulphates               0.487218
chlorides               0.512678
volatile_acidity        0.653036
dtype: float64
```

In [2]:
```
sns.set_context('talk')

sns.set_style('white')
```

In [6]:
```
ax = correlations.plot(kind='bar', color=colors[0])
ax.set(ylim=[-1, 1], ylabel='pearson correlation');
```



In [25]:
```
#we want to do is get the absolute values for each.
correlations.map(abs)
```

Out[25]:
```
total_sulfur_dioxide    0.700357
free_sulfur_dioxide     0.471644
residual_sugar          0.348821
citric_acid             0.187397
quality                 0.119323
alcohol                 0.032970
pH                      0.329129
density                 0.390645
fixed_acidity           0.486740
sulphates               0.487218
chlorides               0.512678
volatile_acidity        0.653036
dtype: float64
```

In [26]:
```
#We can then sort those values by calling sort values.
#we probably just want the last two
correlations.map(abs).sort_values()
```

Out[26]:
```
alcohol                 0.032970
quality                 0.119323
citric_acid             0.187397
pH                      0.329129
residual_sugar          0.348821
density                 0.390645
free_sulfur_dioxide     0.471644
fixed_acidity           0.486740
sulphates               0.487218
chlorides               0.512678
volatile_acidity        0.653036
total_sulfur_dioxide    0.700357
dtype: float64
```

In [29]:
```
correlations.map(abs).sort_values().iloc[-2:]
```

Out[29]:
```
volatile_acidity        0.653036
total_sulfur_dioxide    0.700357
dtype: float64
```

In [30]:
```
correlations.map(abs).sort_values().iloc[-2:].index
```

Out[30]:
```
Index(['volatile_acidity', 'total_sulfur_dioxide'], dtype='object')
```

In [31]:
```
from sklearn.preprocessing import MinMaxScaler

fields = correlations.map(abs).sort_values().iloc[-2:].index
print(fields)
X = data[fields] # X, rather than being all of the fields that we have available, is just going to be equal to these two fields
scaler = MinMaxScaler()
X = scaler.fit_transform(X)
X = pd.DataFrame(X, columns=['%s_scaled' % fld for fld in fields]) #make sure that it goes back into a Pandas DataFrame rather than than a numpy array.
print(X.columns)
```

```
Index(['volatile_acidity', 'total_sulfur_dioxide'], dtype='object')
Index(['volatile_acidity_scaled', 'total_sulfur_dioxide_scaled'], dtype='object')
```

Linear Decision Boundary

Our goal is to look at the decision boundary of a LinearSVC classifier on this dataset. Check out this example in sklearn's documentation.

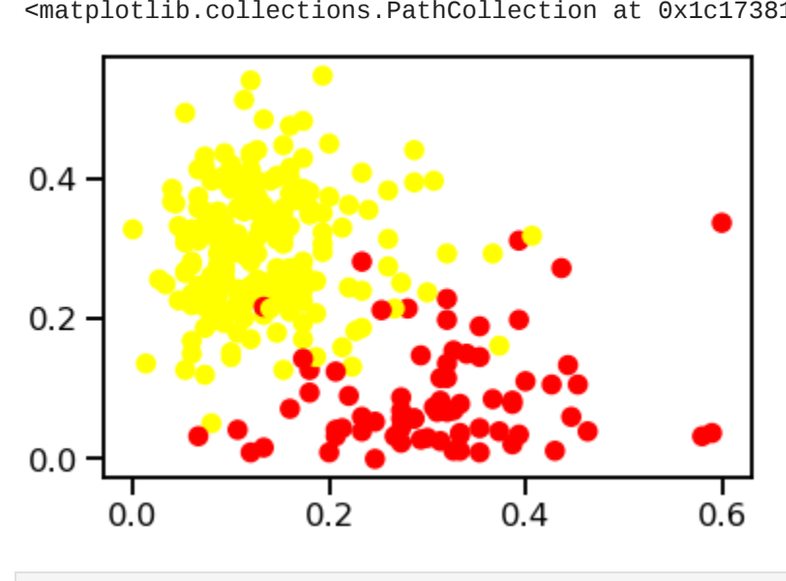
- Fit a Linear Support Vector Machine Classifier to `X`, `y`.
- Pick 300 samples from `X`. Get the corresponding `y` value. Store them in variables `X_color` and `y_color`. This is because original dataset is too large and it produces a crowded plot.
- Modify `y_color` so that it has the value "red" instead of 1 and 'yellow' instead of 0.
- Scatter plot X_color's columns. Use the keyword argument "color=y_color" to color code samples.
- Use the code snippet below to plot the decision surface in a color coded way.

```
x_axis, y_axis = np.arange(0, 1, .005), np.arange(0, 1, .005)
xx, yy = np.meshgrid(x_axis, y_axis)
xx_ravel = xx.ravel()
yy_ravel = yy.ravel()
X_grid = pd.DataFrame([xx_ravel, yy_ravel]).T
y_grid_predictions = *[YOUR MODEL]*.predict(X_grid)
y_grid_predictions = y_grid_predictions.reshape(xx.shape)
ax.contourf(xx, yy, y_grid_predictions, cmap=colors, alpha=.3)
```

With LinearSVC, it is easy to experiment with different parameter choices and see the decision boundary.

In [34]:
```
from sklearn.svm import LinearSVC

LSVC = LinearSVC()
LSVC.fit(X, y)

X_color = X.sample(300, random_state=45)
y_color = y.loc[X_color.index]
y_color = y_color.map(lambda r: 'red' if r == 1 else 'yellow')
ax = plt.axes()
ax.scatter(
    X_color.iloc[:, 0], X_color.iloc[:, 1],
    color=y_color, alpha=1)
#scatter plot without the line seperating two labels
```

Out[34]: <matplotlib.collections.PathCollection at 0x1c17381d760>



In [35]:
```
xx.shape
```

Out[35]:
```
(201, 201)
```

In [36]:
```
yy.shape
```

Out[36]:
```
(201, 201)
```

In [38]:
```
xx.ravel()
```

Out[38]:
```
array([0.   , 0.005, 0.01 , ..., 0.99 , 0.995, 1.   ])
```

In [40]:
```
xx.ravel().shape # very big
```

Out[40]:
```
(40401,)
```

In [39]:
```
yy.ravel()
```

Out[39]:
```
array([0., 0., 0., ..., 1., 1., 1.])
```

In [41]:
```
yy.ravel().shape
```

Out[41]:
```
(40401,)
```

In [42]:
```
pd.DataFrame([xx_ravel, yy_ravel]).T #we put this all into a data frame, and get our grid.
#every single value of y, and each of those x's and y's are meant to represent a fictional amount of either acidity or total sulfur dioxide,
```

Out[42]:

	0	1
0	0.000	0.0
1	0.005	0.0
2	0.010	0.0
3	0.015	0.0
4	0.020	0.0
...	...	...
40396	0.980	1.0
40397	0.985	1.0
40398	0.990	1.0
40399	0.995	1.0
40400	1.000	1.0

40401 rows × 2 columns

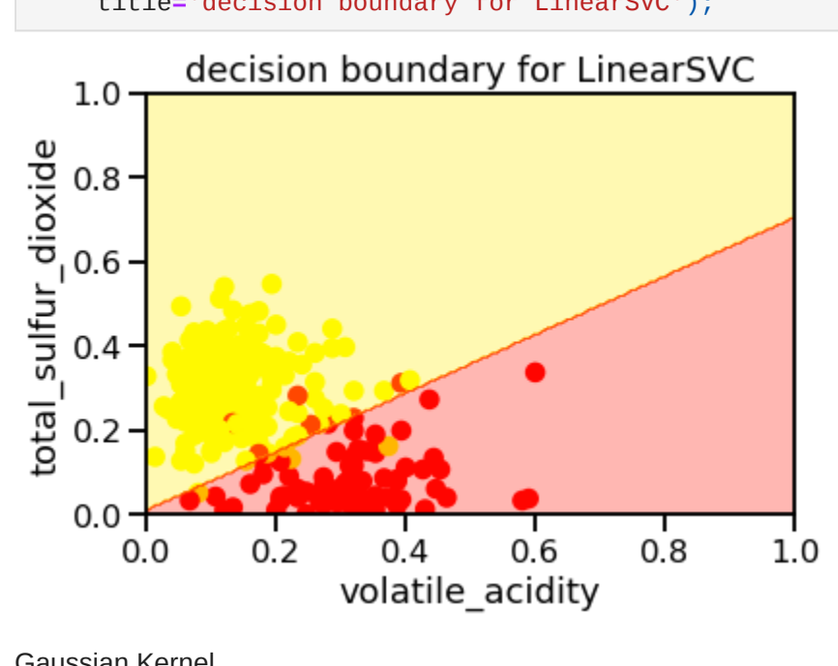
In [43]:
```
LSVC.predict(X_grid) # single array
```

Out[43]:
```
array([1, 1, 1, ..., 0, 0, 0])
```

In [45]:
```
#it's now going to be that same shape as xx as well as yy, and when we run our contour graph, we can call xx, yy.
LSVC.predict(X_grid).reshape(xx.shape)
```

Out[45]:
```
array([[1, 1, 1, ..., 1, 1, 1],
       [1, 1, 1, ..., 1, 1, 1],
       [0, 1, 1, ..., 1, 1, 1],
       ...,
       [0, 0, 0, ..., 0, 0, 0],
       [0, 0, 0, ..., 0, 0, 0],
       [0, 0, 0, ..., 0, 0, 0]])
```

In [48]:
```
from sklearn.svm import LinearSVC

LSVC = LinearSVC()
LSVC.fit(X, y)

X_color = X.sample(300, random_state=45)
y_color = y.loc[X_color.index]
y_color = y_color.map(lambda r: 'red' if r == 1 else 'yellow')
ax = plt.axes()
ax.scatter(
    X_color.iloc[:, 0], X_color.iloc[:, 1],
    color=y_color, alpha=1)
#scatter plot without the line seperating two labels
# -----------
x_axis, y_axis = np.arange(0, 1.005, .005), np.arange(0, 1.005, .005)
xx, yy = np.meshgrid(x_axis, y_axis)
xx_ravel = xx.ravel()
yy_ravel = yy.ravel()
X_grid = pd.DataFrame([xx_ravel, yy_ravel]).T #we put this all into a data frame, and get our grid.
y_grid_predictions = LSVC.predict(X_grid)
y_grid_predictions = y_grid_predictions.reshape(xx.shape)
ax.contourf(xx, yy, y_grid_predictions, cmap=plt.cm.autumn_r, alpha=.3)
# -----------
ax.set(
    xlabel=fields[0],
    ylabel=fields[1],
    xlim=[0, 1],
    ylim=[0, 1],
    title='decision boundary for LinearSVC');
```



Gaussian Kernel

Let's now fit a Gaussian kernel SVC and see how the decision boundary changes.

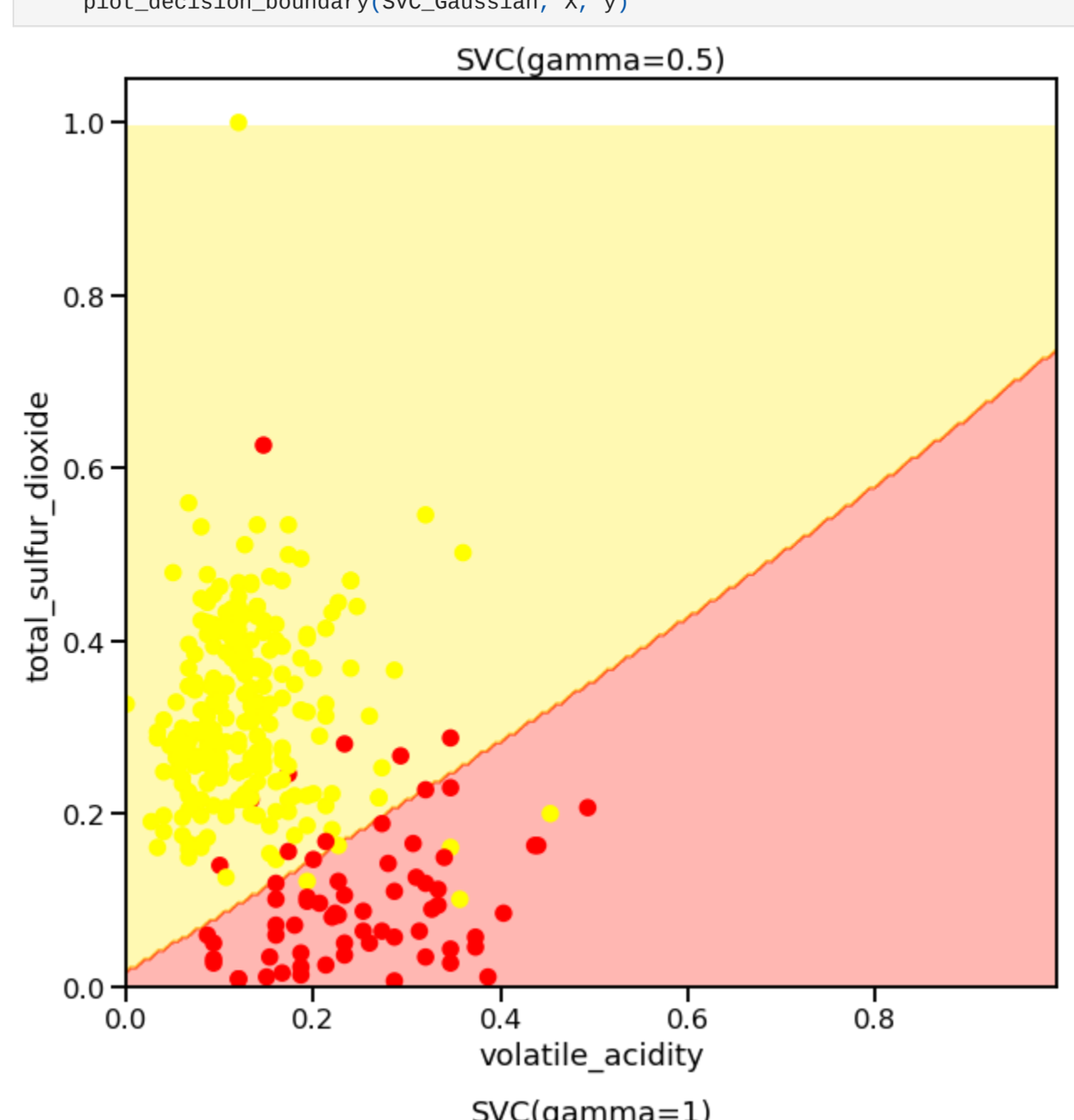
- Consolidate the code snippets in Question 2 into one function which takes in an estimator, `X` and `y`, and produces the final plot with decision boundary. The steps are:
  1. fit model
  2. get sample 300 records from X and the corresponding y's
  3. create grid, predict, plot using ax.contourf
  4. add on the scatter plot
- After copying and pasting code, the finished function uses the input `estimator` and not the LinearSVC model.
- For the following values of `gamma`, create a Gaussian Kernel SVC and plot the decision boundary.
  `gammas = [.5, 1, 2, 10]`
- Holding `gamma` constant, we plot the decision boundary for various values of `C`: `[.1, 1, 10]`

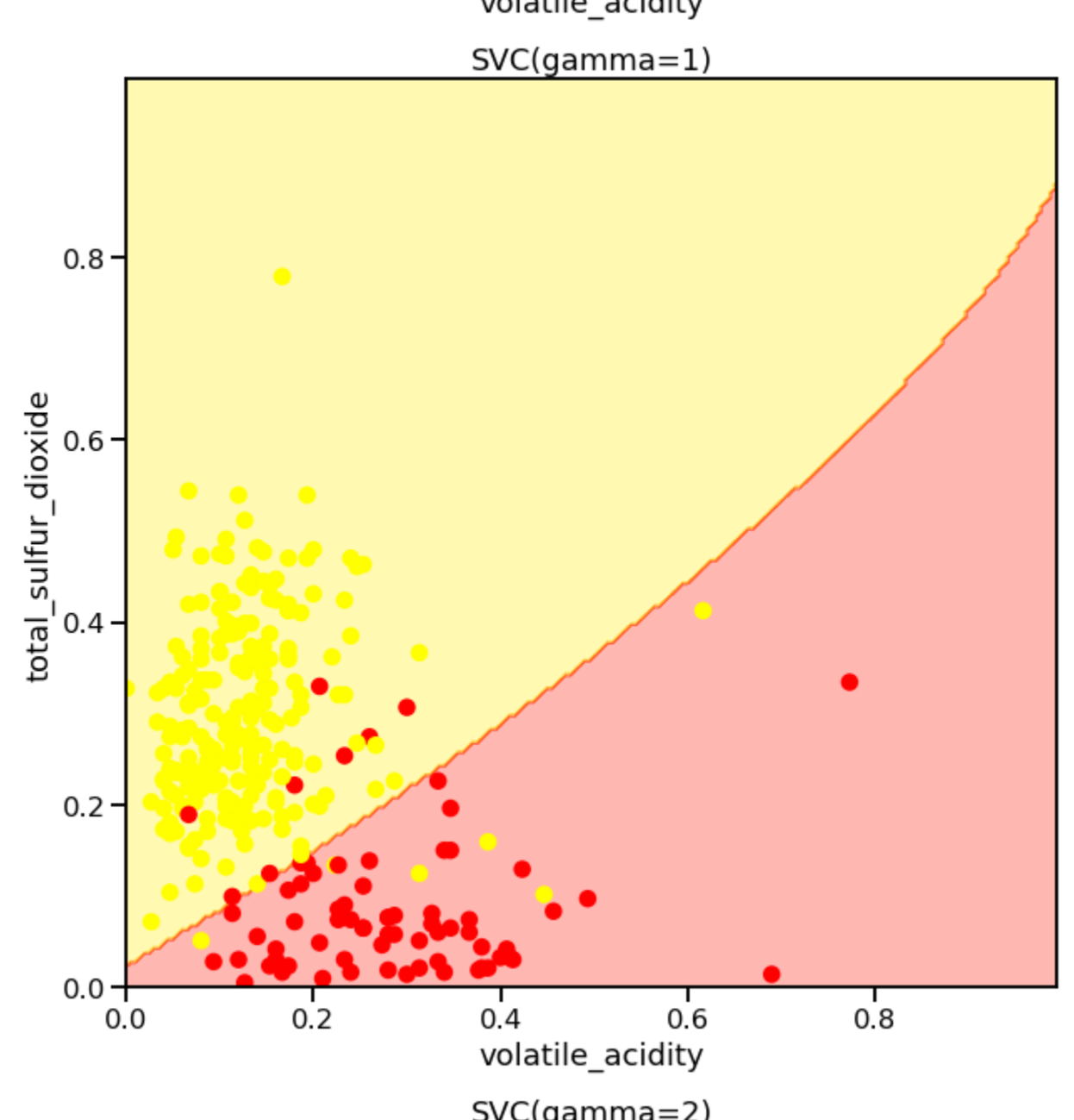
In [49]:
```
def plot_decision_boundary(estimator, X, y):
    estimator.fit(X, y)
    X_color = X.sample(300)
    y_color = y.loc[X_color.index]
    y_color = y_color.map(lambda r: 'red' if r == 1 else 'yellow')
    x_axis, y_axis = np.arange(0, 1, .005), np.arange(0, 1, .005)
    xx, yy = np.meshgrid(x_axis, y_axis)
    xx_ravel = xx.ravel()
    yy_ravel = yy.ravel()
    X_grid = pd.DataFrame([xx_ravel, yy_ravel]).T
    y_grid_predictions = estimator.predict(X_grid)
    y_grid_predictions = y_grid_predictions.reshape(xx.shape)

    fig, ax = plt.subplots(figsize=(10, 10))
    ax.contourf(xx, yy, y_grid_predictions, cmap=plt.cm.autumn_r, alpha=.3)
    ax.scatter(X_color.iloc[:, 0], X_color.iloc[:, 1], color=y_color, alpha=1)
    ax.set(
        xlabel=fields[0],
        ylabel=fields[1],
        title=str(estimator))
```

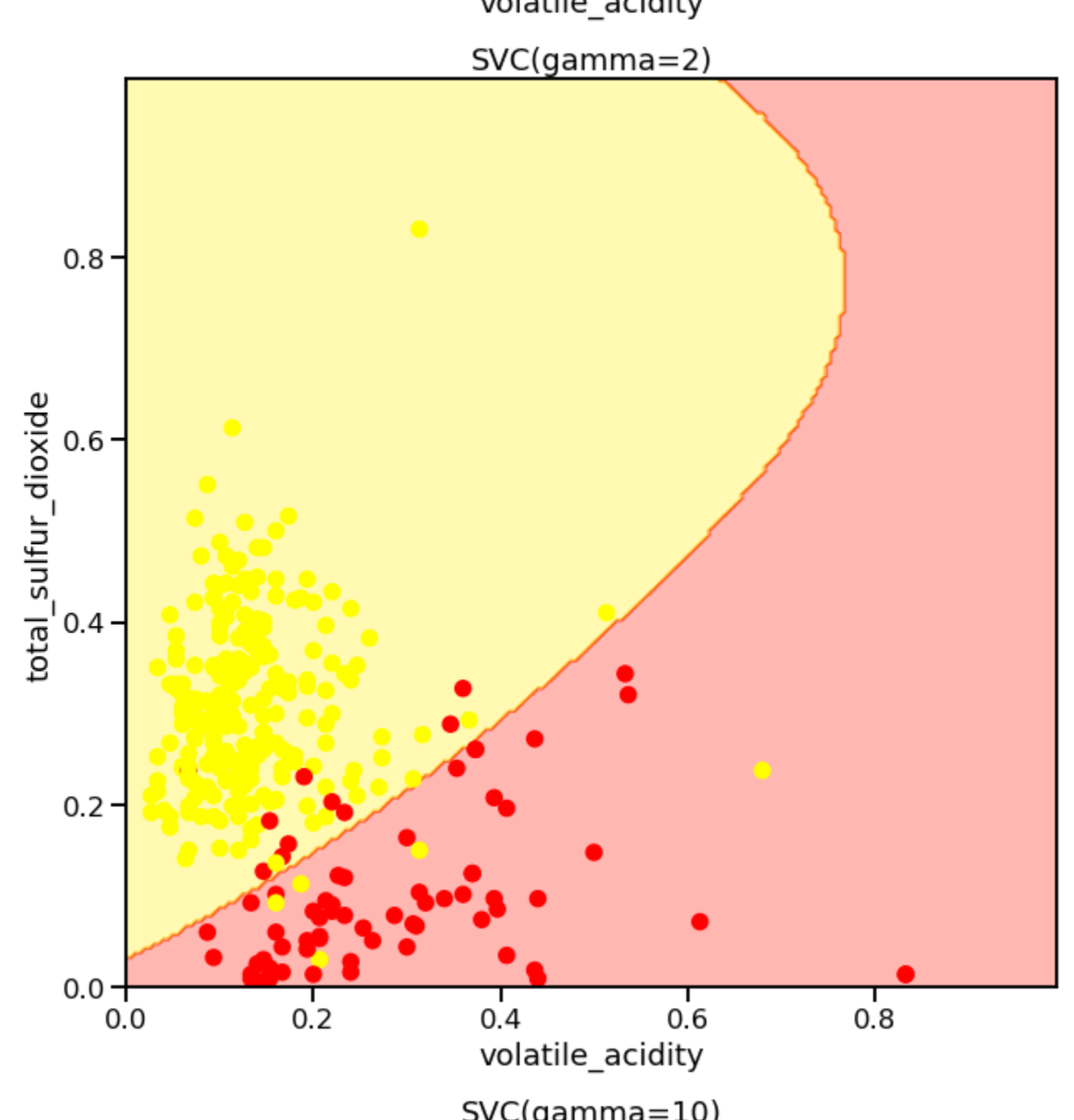
So we're going to create that function called plot_decision_boundary. Then we're going to set gamma is equal to a list of four different gammas that we're going to run here. For each one of those gammas, we're going to first initiate an SVC model called SVC Gaussian. The kernel that we're going to use is going to be that radial basis function and the gamma value is going to be first 0.5, then 1, then 2, then 10. For each one, we're going to use that plot_decision_boundary function that we just defined above and plot out each one of the decision boundaries.

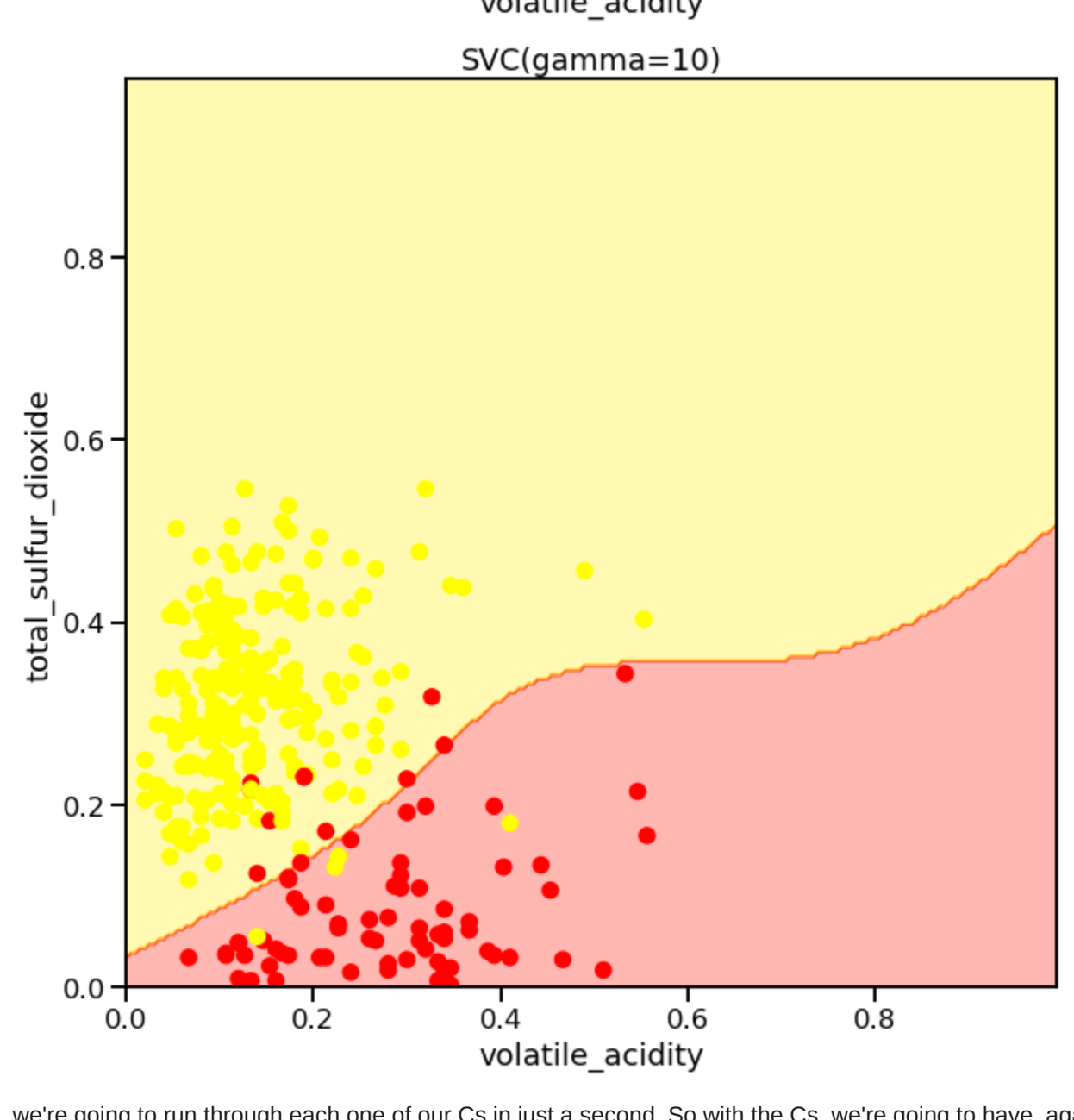
If you'll look at line two here of SVC, we see that gamma is equal to 0.5 towards the end of that line, and we see that there's pretty high regularization. We have a pretty non-complex model. Remember, high regularization means a not so complex model. We then increase gamma to one and we see a bit more curvature in our line. So it's a bit more of a complex model. Then when gamma is equal to two, again increasing gamma, so we're reducing regularization by increasing gamma, and therefore coming up with a more complex model because there's less regularization. We have even more curvature. Then when we say gamma equal to 10, we see this wavy decision boundary that's getting closer towards over-fitting. Perhaps, this is still a decent decision boundary. But as we keep increasing gamma, we see that we increase the complexity because we're reducing the penalty that we'd have for our regularization terms.

In [50]:
```
from sklearn.svm import SVC

gammas = [.5, 1, 2, 10]
for gamma in gammas:
    SVC_Gaussian = SVC(kernel='rbf', gamma=gamma)
    plot_decision_boundary(SVC_Gaussian, X, y)
```





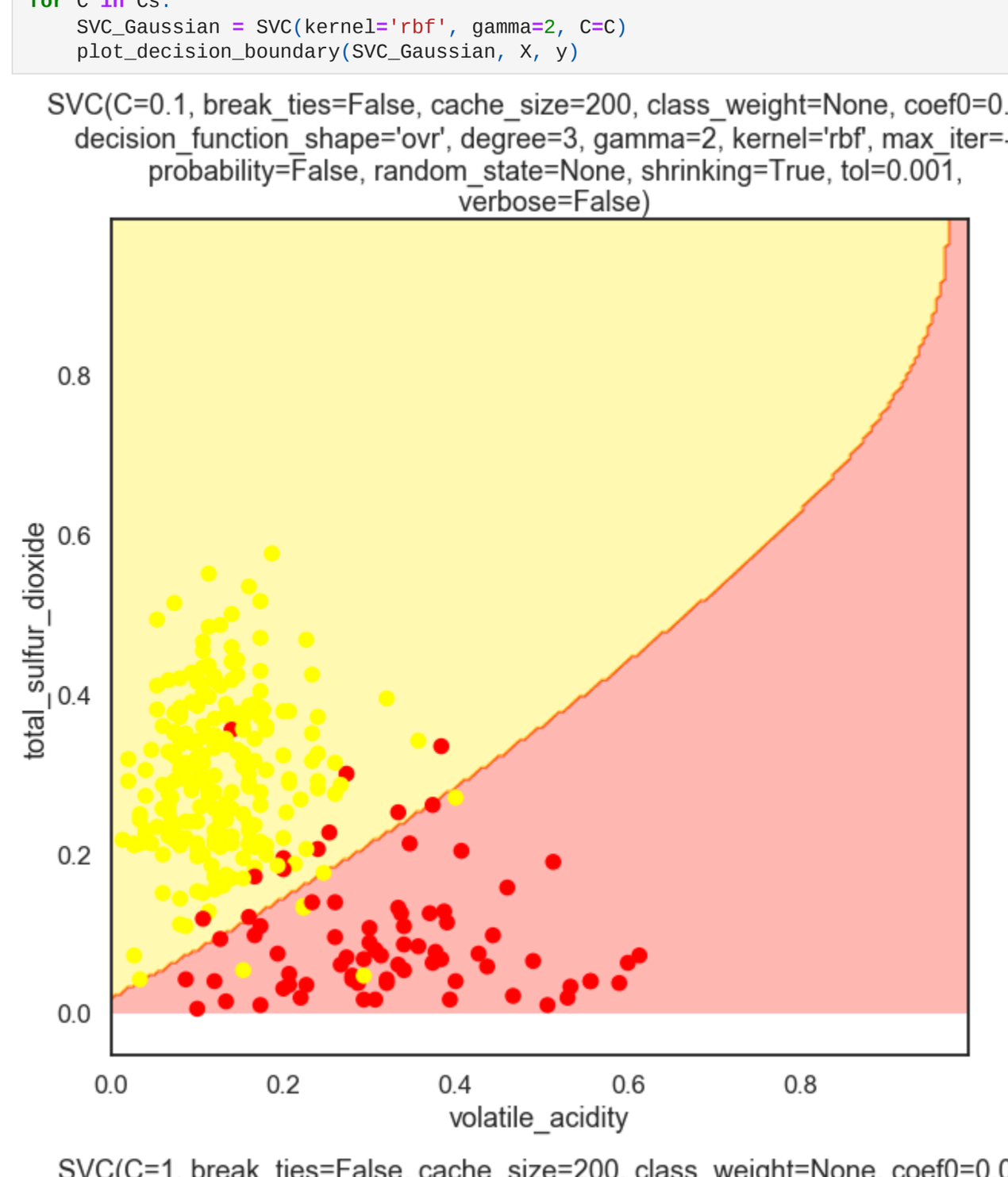


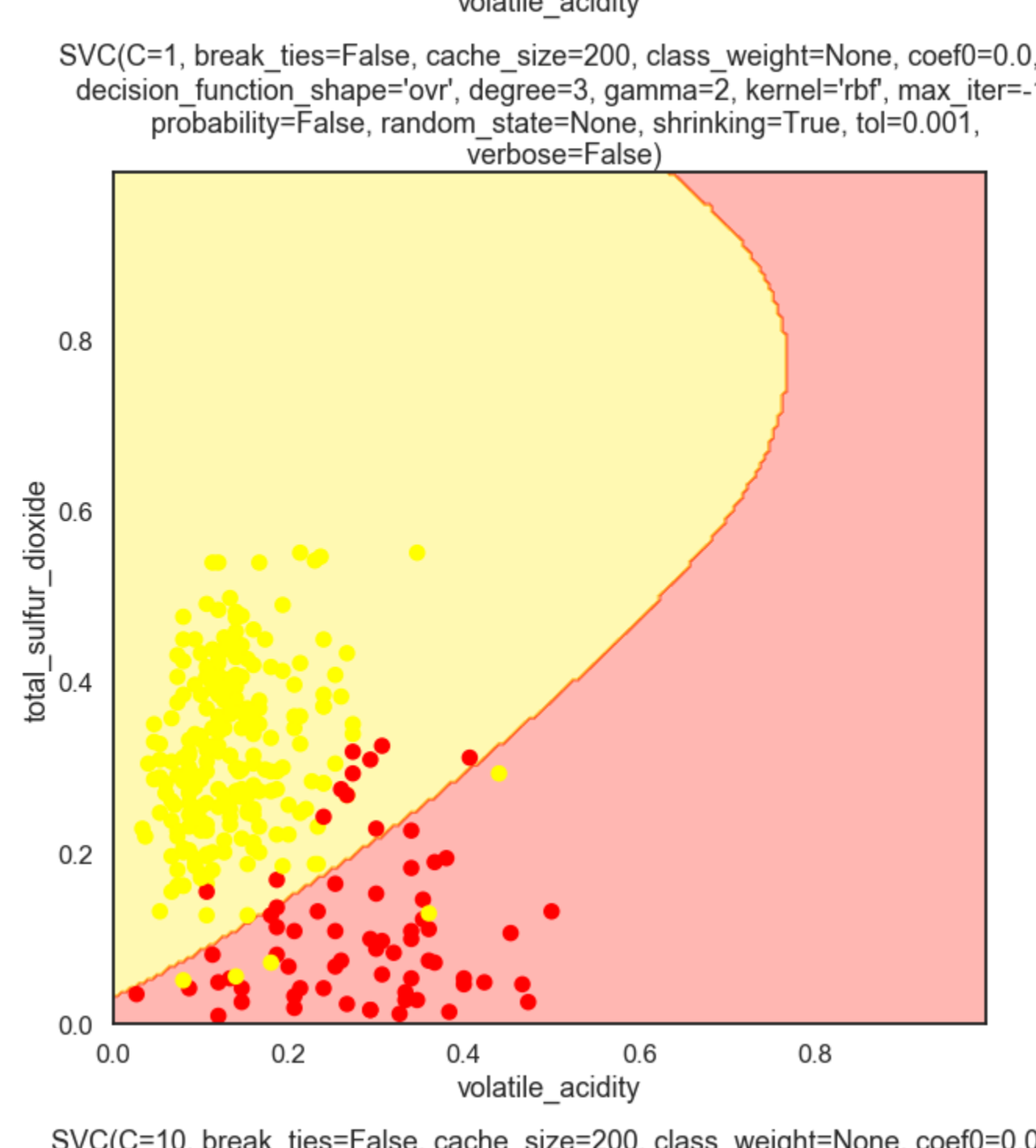


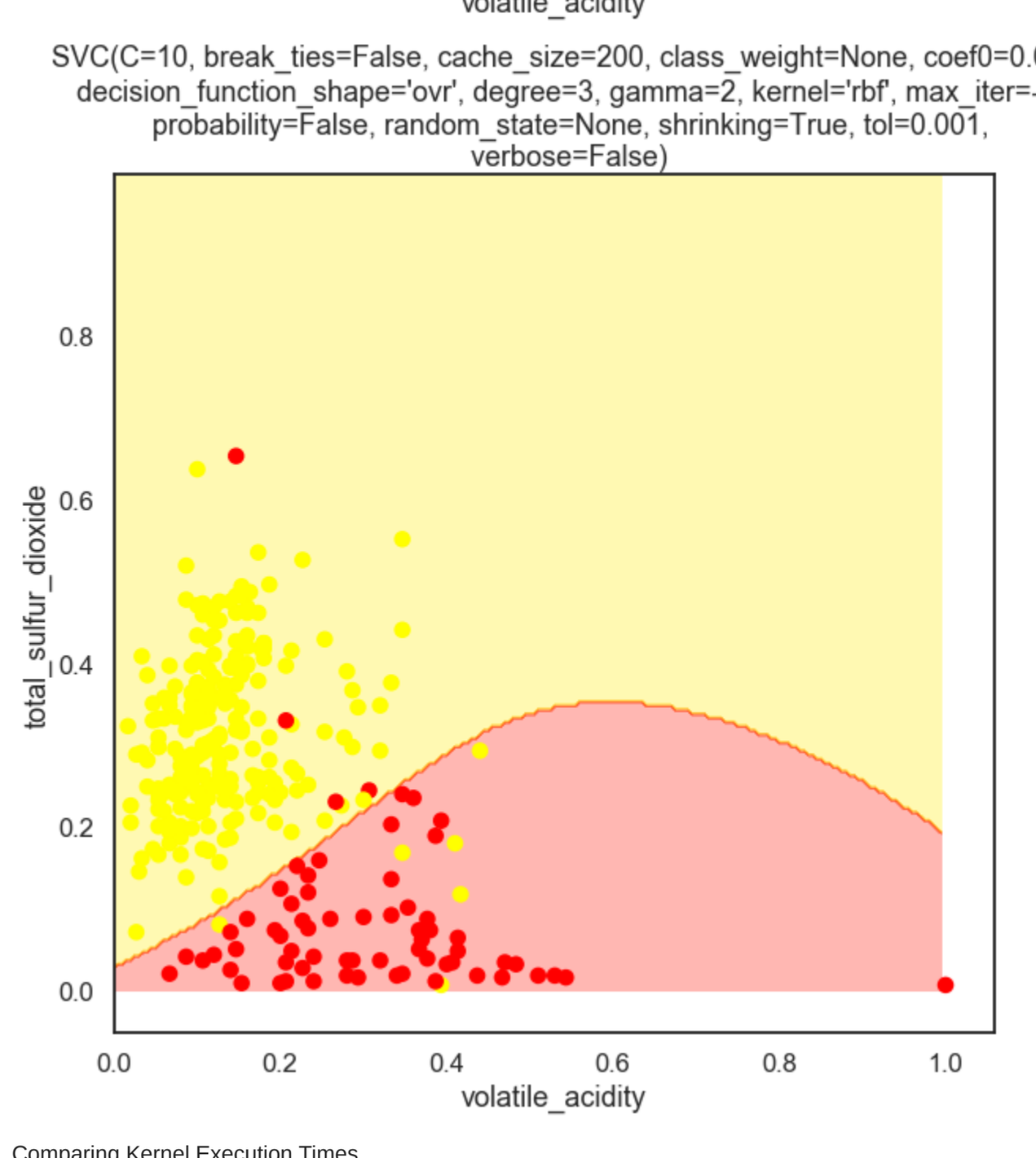
we're going to run through each one of our Cs in just a second. So with the Cs, we're going to have, again, increasing from 0.1, to 1, to 10, and we're going to call SVC. We're going to say gamma equal to a constant, here being two, then we're going to loop through each one of these different C values. Again, for that I want you to start thinking which ones are going to have higher versus lower values in regards to the amount of regularization. So remember, more regularization for, I want you to think for higher value of C versus lower value of C, as well as think about the complexity of the model once you reduce or increase the regularization term.

Then the same holds for C. So it's the same inverse relationship where the lower value of C, so here we're starting off with C equals 0.1, lower value of C means that we're going to have higher regularization. Higher regularization means a less complex model. Then as we increase C, we see how it gets more complex, more curvature to our model.

In [11]:
```
Cs = [.1, 1, 10]
for C in Cs:
    SVC_Gaussian = SVC(kernel='rbf', gamma=2, C=C)
    plot_decision_boundary(SVC_Gaussian, X, y)
```







Comparing Kernel Execution Times

In this exercise, we will compare the fitting times between SVC vs Nystroem with rbf kernel.

Jupyter Notebooks provide a useful magic function `%timeit` which executes a line and prints out the time it took to fit. If we type `%%timeit` in the beginning of the cell, it will output the execution time.

We proceed with the following steps:

- Create `y` from data.color, and `X` from the rest of the columns.
- Use `%%timeit` to get the time for fitting an SVC with rbf kernel.
- Use `%%timeit` to get the time for the following: fit_transform the data with Nystroem and then fit a SGDClassifier.

Nystroem+SGD will take much shorter to fit. This difference will be more pronounced if the dataset was bigger.

- Make 5 copies of X and concatenate them
- Make 5 copies of y and concatenate them
- Compare the time it takes to fit the both methods above

In [51]:
```
from sklearn.kernel_approximation import Nystroem
from sklearn.svm import SVC
from sklearn.linear_model import SGDClassifier

y = data.color == 'red'
X = data[data.columns[:-1]]

kwargs = {'kernel': 'rbf'} #kwargs, and we can parse that in as just a something that you pass him within Python. As long as you pass in a dictionary, you're saying for the kernel
svc = SVC(**kwargs)
nystroem = Nystroem(**kwargs)
sgd = SGDClassifier()
```

In [52]:
```
%%timeit
svc.fit(X, y)
```

```
632 ms ± 40.5 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

In [54]:
```
X
```

Out[54]:

	fixed_acidity	volatile_acidity	citric_acid	residual_sugar	chlorides	free_sulfur_dioxide	total_sulfur_dioxide	density	pH	sulphates	alcohol	quality
0	7.4	0.70	0.00	1.9	0.076	11.0	34.0	0.99780	3.51	0.56	9.4	5
1	7.8	0.88	0.00	2.6	0.098	25.0	67.0	0.99680	3.20	0.68	9.8	5
2	7.8	0.76	0.04	2.3	0.092	15.0	54.0	0.99700	3.26	0.65	9.8	5
3	11.2	0.28	0.56	1.9	0.075	17.0	60.0	0.99800	3.16	0.58	9.8	6
4	7.4	0.70	0.00	1.9	0.076	11.0	34.0	0.99780	3.51	0.56	9.4	5
...	...	...	...	...	...	...	...	...	...	...	...	...
6492	6.2	0.21	0.29	1.6	0.039	24.0	92.0	0.99114	3.27	0.50	11.2	6
6493	6.6	0.32	0.36	8.0	0.047	57.0	168.0	0.99490	3.15	0.46	9.6	5
6494	6.5	0.24	0.19	1.2	0.041	30.0	111.0	0.99254	2.99	0.46	9.4	6
6495	5.5	0.29	0.30	1.1	0.022	20.0	110.0	0.98869	3.34	0.38	12.8	7
6496	6.0	0.21	0.38	0.8	0.020	22.0	98.0	0.98941	3.26	0.32	11.8	6

6497 rows × 12 columns