Perspective in Computational Modeling

Homework 6
Tianyue Niu

In [41]:
```
#import necessary packages
import random
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sb
import math
from sklearn import svm
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import GridSearchCV

#disable future warning
import warnings
warnings.simplefilter(action='ignore', category=FutureWarning)
```

Conceptual Question

1.(15 points) Generate a simulated two-class data set with 100 observations and two features in which there is a visible (clear) but still non-linear separation between the two classes. Show that in this setting, a support vector machine with a radial kernel will outperform a support vector classifier (a linear kernel) on the training data. Which technique performs best on the test data? Make plots and report training and test error rates in order to support your conclusions.

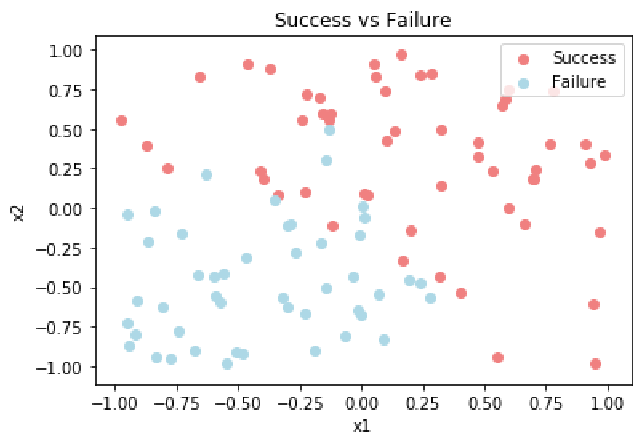
In [7]:
```
np.random.seed(2)
#create predictors x1 & x2
x1 = np.random.uniform(-1,1,100)
x2 = np.random.uniform(-1,1,100)

#create error term
e = np.random.normal(0, 0.5, 100)

#calculate Y
y = x1 + x1**2 +x1**3 + x2+ x2**2 + x2**3 + e
prob_success = math.e**y/(1+math.e**y)

success = prob_success > 0.5 #create boolean array for success
failure = prob_success <= 0.5 #create boolean array for failures

#plot x1 and x2
plt.scatter(x1[success], x2[success], color = 'lightcoral')
plt.scatter(x1[failure], x2[failure], color = 'lightblue')
plt.xlabel('x1')
plt.ylabel('x2')
plt.title('Success vs Failure')
plt.legend(['Success', 'Failure'], loc=1);
```



In [8]:
```
X = np.column_stack((x1,x2)) #stack x1 and x2 together
X = pd.DataFrame(X) #convert to pandas data frame
```

In [9]:
```
X_train, X_test, y_train, y_test = train_test_split(X, success, test_size=0.2, random_state=42)
```

In [10]:
```
svm_radial = svm.SVC().fit(X_train, y_train)
```

In [11]:
```
svm_linear = svm.SVC(kernel='linear').fit(X_train, y_train)
```

In [12]:
```
print("SVM with radial kerneling has an accuracy of:",
      svm_radial.score(X_train, y_train), "for training data set.")
print("SVM with a linear kernel has an accuracy of:",
      svm_linear.score(X_train, y_train), "for training data set.")
```

```
SVM with radial kerneling has an accuracy of: 0.9125 for training data set.
SVM with a linear kernel has an accuracy of: 0.8875 for training data set.
```

In [13]:
```
print("SVM with radial kerneling has an accuracy of:",
      svm_radial.score(X_test, y_test), "for testing data set.")
print("SVM with a linear kernel has an accuracy of:",
      svm_linear.score(X_test, y_test), "for testing data set.")
```

```
SVM with radial kerneling has an accuracy of: 0.8 for testing data set.
SVM with a linear kernel has an accuracy of: 0.75 for testing data set.
```

```
We see that SVM with radial kerneling performs better than linear kernel SVM in both training and
 testing data sets.
```

SVM vs logistic regression

2.(5 points) Generate a data set with n = 500 and p = 2, such that the observations belong to two classes with some overlapping, non-linear boundary between them.

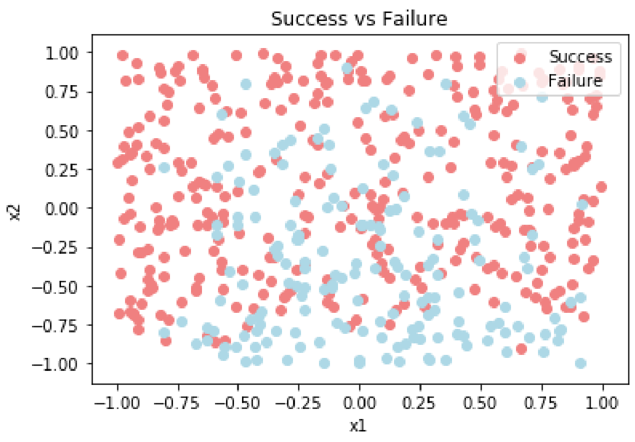
In [14]:
```
np.random.seed(2)
x1 = np.random.uniform(-1,1,500)
x2 = np.random.uniform(-1,1,500)
e = np.random.normal(0, 0.5, 500)

#calculate Y
y = x1**2 + x2**3 + e
prob_success = math.e**y/(1+math.e**y)

success = prob_success > 0.5 #create boolean array for success
failure = prob_success <= 0.5 #create boolean array for failures
```

3.(5 points) Plot the observations with colors according to their class labels (y). Your plot should display X1 on the x-axis and X2 on the y-axis.

In [15]:
```
plt.scatter(x1[success], x2[success], color = 'lightcoral')
plt.scatter(x1[failure], x2[failure], color = 'lightblue')
plt.xlabel('x1')
plt.ylabel('x2')
plt.title('Success vs Failure')
plt.legend(['Success', 'Failure'], loc=1);
```



4.(5 points) Fit a logistic regression model to the data, using X1 and X2 as predictors.

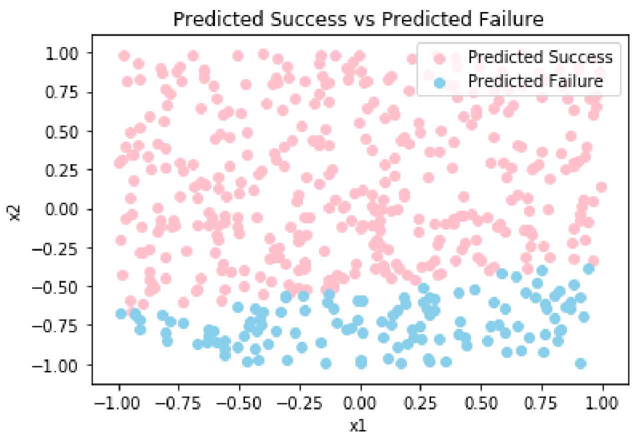
In [16]:
```
X = np.column_stack((x1,x2)) #stack x1 and x2 together
X = pd.DataFrame(X) #convert to pandas data frame
```

In [17]:
```
lr = LogisticRegression().fit(X, success)
```

5.(5 points) Obtain a predicted class label for each observation based on the logistic model previously fit. Plot the observations, colored according to the predicted class labels (the predicted decision boundary should look linear).

In [18]:
```
lr_pred = lr.predict(X)
```

In [19]:
```
plt.scatter(x1[lr_pred], x2[lr_pred], color = 'pink')
plt.scatter(x1[~lr_pred], x2[~lr_pred], color = 'skyblue')
plt.xlabel('x1')
plt.ylabel('x2')
plt.title('Predicted Success vs Predicted Failure')
plt.legend(['Predicted Success', 'Predicted Failure'], loc=1);
```



```
The decision boundary clearly looks linear.
```

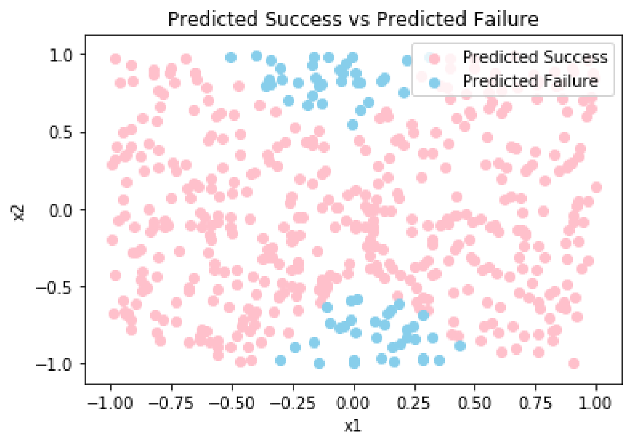
6.(5 points) Now fit a logistic regression model to the data, but this time using some non-linear function of both X1 and X2 as predictors (e.g. X12, X1 × X2, log(X2), and so on).

In [20]:
```
non_linear_p1 = x1*x2
non_linear_p2 = x1**2
non_linear_p3 = x2**2
X_non_linear = np.column_stack((non_linear_p1,non_linear_p2,non_linear_p3))
lr_non_linear = LogisticRegression().fit(X_non_linear, success)
```

7.(5 points) Now, obtain a predicted class label for each observation based on the fitted model with non-linear transformations of the X features in the previous question. Plot the observations, colored according to the new predicted class labels from the non-linear model (the decision boundary should now be obviously non-linear). If it is not, then repeat earlier steps until you come up with an example in which the predicted class labels and the resultant decision boundary are clearly non-linear.

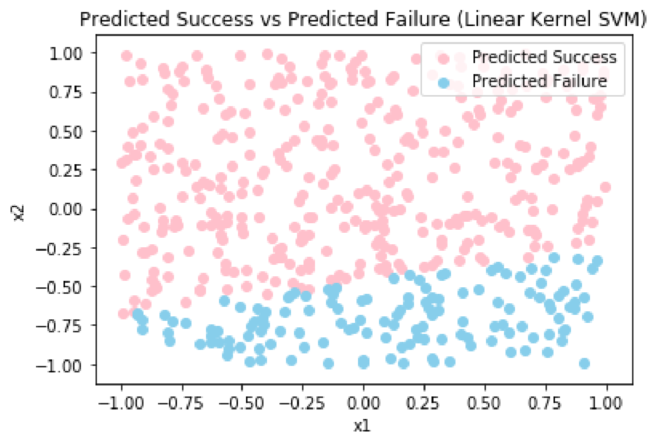
In [21]:
```
lr_pred_non_linear = lr_non_linear.predict(X_non_linear)
```

In [22]:
```
plt.scatter(x1[lr_pred_non_linear], x2[lr_pred_non_linear], color = 'pink')
plt.scatter(x1[~lr_pred_non_linear], x2[~lr_pred_non_linear], color = 'skyblue')
plt.xlabel('x1')
plt.ylabel('x2')
plt.title('Predicted Success vs Predicted Failure')
plt.legend(['Predicted Success', 'Predicted Failure'], loc=1);
```



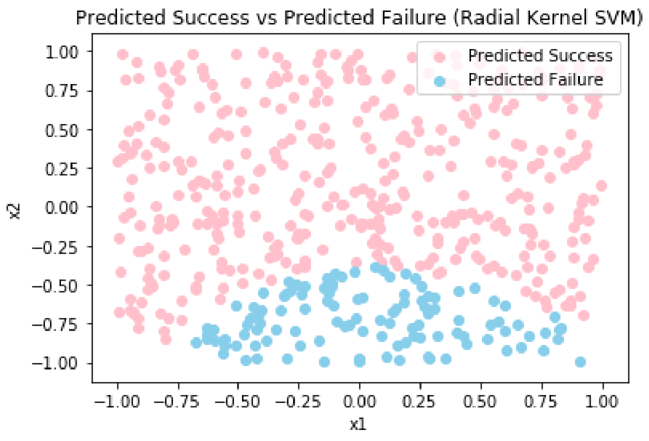
8.(5 points) Now, fit a support vector classifier (linear kernel) to the data with original X1 and X2 as predictors. Obtain a class prediction for each observation. Plot the observations, colored according to the predicted class labels.

In [23]:
```
svm_linear = svm.SVC(kernel='linear').fit(X, success)
svm_pred = svm_linear.predict(X)
plt.scatter(x1[svm_pred], x2[svm_pred], color = 'pink')
plt.scatter(x1[~svm_pred], x2[~svm_pred], color = 'skyblue')
plt.xlabel('x1')
plt.ylabel('x2')
plt.title('Predicted Success vs Predicted Failure (Linear Kernel SVM)')
plt.legend(['Predicted Success', 'Predicted Failure'], loc=1);
```



9.(5 points) Fit a SVM using a non-linear kernel to the data. Obtain a class prediction for each observation. Plot the observations, colored according to the predicted class labels.

In [24]:
```
svm_r = svm.SVC().fit(X, success)
svm_pred_r = svm_r.predict(X)
plt.scatter(x1[svm_pred_r], x2[svm_pred_r], color = 'pink')
plt.scatter(x1[~svm_pred_r], x2[~svm_pred_r], color = 'skyblue')
plt.xlabel('x1')
plt.ylabel('x2')
plt.title('Predicted Success vs Predicted Failure (Radial Kernel SVM)')
plt.legend(['Predicted Success', 'Predicted Failure'], loc=1);
```



10.(5 points) Discuss your results and specifically the tradeoffs between estimating non-linear decision boundaries using these two different approaches.

In [25]:
```
print('Logistic regression with X1, X2 as predictors:', lr.score(X, success))
print('Logistic regression with non-linear function as predictors:',
      lr_non_linear.score(X_non_linear, success))
print('SVM linear kernel:', svm_linear.score(X, success))
print('SVM radial kernel', svm_r.score(X, success))
```

```
Logistic regression with X1, X2 as predictors: 0.722
Logistic regression with non-linear function as predictors: 0.644
SVM linear kernel: 0.718
SVM radial kernel 0.768
```

```
Logistic regression with non-linear functions of X1 & X2 as predictors performed worse. This might
be that the non-linear functions of X1 & X2 is actually not a good representation of the underlyin
g functional form.

We see that in this case, SVM with radial kernel performed better than the other classifiers. In c
ase of non-linear relationship, SVM with radial kerneling should be preferred over linear kernelin
g.
```

Tuning Cost

11.(5 points) Generate two-class data with p = 2 in such a way that the classes are just barely linearly separable.

In [26]:
```
np.random.seed(2)
x1 = np.random.uniform(-1,1,500)
x2 = np.random.uniform(-1,1,500)
e = np.random.normal(0, 0.5, 500)

#calculate Y
y = x1 + x2+ e
prob_success = math.e**y/(1+math.e**y)

#stack x together
X = np.column_stack((x1,x2)) #stack x1 and x2 together
X = pd.DataFrame(X) #convert to pandas data frame

success = prob_success > 0.5 #create boolean array for success
failure = prob_success <= 0.5 #create boolean array for failures
```

In [27]:
```
X_train, X_test, y_train, y_test = train_test_split(X, success, test_size=0.2, random_state=42)
```

In [46]:
```
def find_best_C(X_train, y_train):
    min_error = 10000
    best_model = None
    best_C = None
    training_errors=[]
    CV_errors = []

    for c in [0.001,0.01,0.1,1,2,3,4,5,6,7,8,9,10]:
        model = svm.SVC(C = c, random_state=666)
        CV_error = 1-np.mean(cross_val_score(model, X_train, y_train, cv=10, scoring='accuracy'
))
        CV_errors.append(CV_error)
        training_errors.append(1-model.fit(X_train,y_train).score(X_train,y_train))
        if CV_error < min_error:
            min_error = CV_error
            best_model = model
            best_C = c

    return best_model, best_C, min_error, training_errors, CV_errors
```

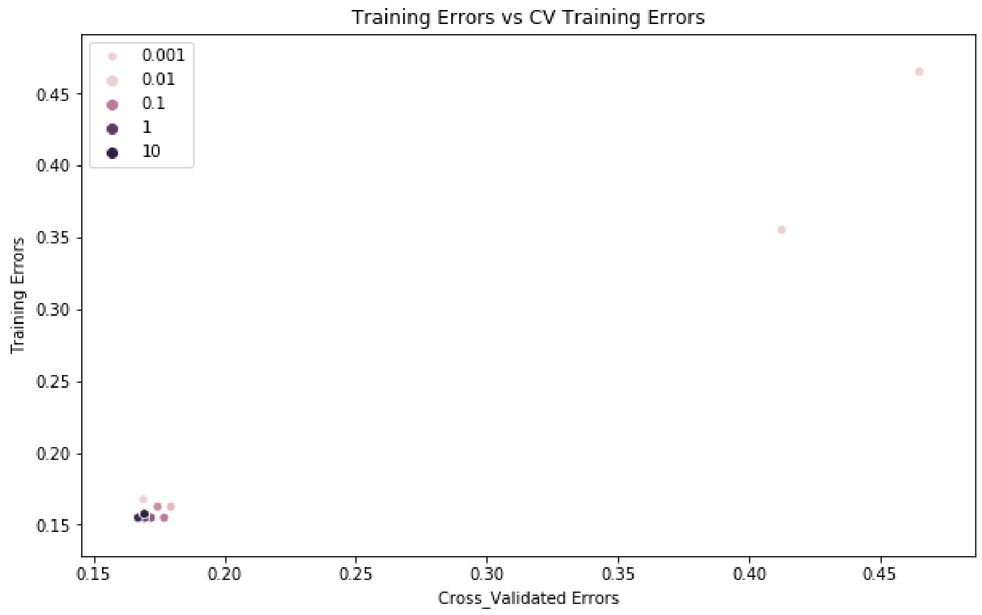
In [29]:
```
best_model, best_C, min_CV_error,training_errors, CV_errors = find_best_C(X_train, y_train)
```

In [30]:
```
print("CV errors:", CV_errors, "\n"*2, "Training errors", training_errors)
```

```
CV errors: [0.4649812382739211, 0.41245309568480304, 0.169029080675422, 0.17947154471544713,
0.17184333958724207, 0.17453252032520328, 0.17697154471544718, 0.17196841776110072, 0.1719684
1776110072, 0.16952939337085682, 0.16952939337085682, 0.16696529080675426, 0.1694043151969980
6]

 Training errors [0.46499999999999997, 0.355, 0.16749999999999998, 0.16249999999999998, 0.155
00000000000003, 0.16249999999999998, 0.15500000000000003, 0.15500000000000003, 0.155000000000
00003, 0.15749999999999997, 0.15500000000000003, 0.15500000000000003, 0.15749999999999997]
```

In [31]:
```
plt.figure(figsize=(10,6))
sb.scatterplot(x = CV_errors, y= training_errors, hue =[0.001,0.01,0.1,1,2,3,4,5,6,7,8,9,10])
plt.legend([0.001,0.01, 0.1,1,10])
plt.ylabel('Training Errors')
plt.xlabel('Cross_Validated Errors')
plt.title('Training Errors vs CV Training Errors');
```



In [32]:
```
print("The cross-validated error is lowest at C=", best_C)
print("The training error is lowest at C=", [0.001,0.01,0.1,1,10][training_errors.index(min(tra
ining_errors))])
```

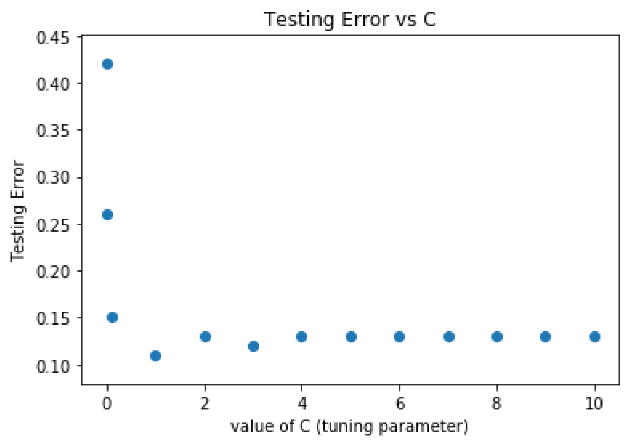
```
The cross-validated error is lowest at C= 9
The training error is lowest at C= 10
```

The above graph shows the relationship between training errors and cross-validated errors for each value of C (from 0.001 to 10). Interestingly, most CV/training errors are relatively small/clustered together when the tuning parameter is larger than 0.01. If we examine the training/CV errors closer, we would see that the errors at C= 0.1/1 and the errors at C=9/10 are very similar to each other.

13.(5 points) Generate an appropriate test data set, and compute the test errors corresponding to each of the values of cost considered. Which value of cost leads to the fewest test errors, and how does this compare to the values of cost that yield the fewest training errors and the fewest cross-validation errors?

In [33]:
```
test_errors_svm = []
for c in [0.001,0.01,0.1,1,2,3,4,5,6,7,8,9,10]:
        model = svm.SVC(C = c, random_state=666).fit(X_train, y_train)
        error = 1 - model.score(X_test, y_test)
        test_errors_svm.append(error)
```

In [34]:
```
plt.scatter([0.001,0.01,0.1,1,2,3,4,5,6,7,8,9,10], test_errors_svm)
plt.ylabel('Testing Error')
plt.xlabel('value of C (tuning parameter)')
plt.title('Testing Error vs C');
```



In [35]:
```
print("Lowest test error occurs when C=",
      [0.001,0.01,0.1,1,10][test_errors_svm.index(min(test_errors_svm))])
```

```
Lowest test error occurs when C= 1
```

14.(5 points) Discuss your results.

```
The above results probably suggest that, when the underlying relationship between our predictors &
the response value is almost linear, we don't need a high C/cost tuning parameter to severely puni
sh violations to the margin. Instead, a low C (around 1) would be enough, as suggested by the lowe
st test error graph.
```

Applications

In [36]:
```
train = pd.read_csv('gss_train.csv')
test = pd.read_csv('gss_test.csv')
X_train_gss = train.drop(columns="colrac")
y_train_gss = train["colrac"]
```

15.(5 points) Fit a support vector classifier to predict colrac as a function of all available predictors, using 10-fold cross-validation to find an optimal value for cost. Report the CV errors associated with different values of cost, and discuss your results.

In [47]:
```
#values of cost evaluated are: [0.001,0.01,0.1,1,2,3,4,5,6,7,8,9,10]
best_model, best_C, min_CV_error,training_errors, CV_errors = find_best_C(X_train_gss, y_train_
gss)
print("Best tuning parameter:", best_C, ", with CV error = ", min_CV_error,
      "\n"*2, "Training Errors:", training_errors,
      "\n"*2, "CV Errors:", CV_errors,)
```

```
Best tuning parameter: 2 , with CV error =  0.2620722386542138

 Training Errors: [0.474679270762998, 0.474679270762998, 0.4659014179608373, 0.02633355840648
2105, 0.002025658338960179, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]

 CV Errors: [0.4746776018588099, 0.4746776018588099, 0.47399732974996645, 0.2728332907623415
3, 0.2620722386542138, 0.26546422124082714, 0.26411280819238636, 0.26411280819238636, 0.26411
280819238636, 0.26411280819238636, 0.26411280819238636, 0.26411280819238636, 0.26411280819238
636]
```

```
The best tuning parameter C that gives the lowest CV error is C=2.
```

16.(15 points) Repeat the previous question, but this time using SVMs with radial and polynomial basis kernels, with different values for gamma and degree and cost. Present and discuss your results (e.g., fit, compare kernels, cost, substantive conclusions across fits, etc.).

```
In [50]: parameters = {'kernel':('poly', 'rbf', 'linear'), 'C':[0.1, 1, 10], 'gamma':('scale','auto')}
```

```
In [51]: svc = svm.SVC()
         model = GridSearchCV(svc, parameters, cv=10)
```

```
In [52]: model.fit(X_train_gss, y_train_gss)
```

```
Out[52]: GridSearchCV(cv=10, error_score='raise-deprecating',
                      estimator=SVC(C=1.0, cache_size=200, class_weight=None, coef0=0.0,
                                    decision_function_shape='ovr', degree=3,
                                    gamma='auto_deprecated', kernel='rbf', max_iter=-1,
                                    probability=False, random_state=None, shrinking=True,
                                    tol=0.001, verbose=False),
                      iid='warn', n_jobs=None,
                      param_grid={'C': [0.1, 1, 10], 'gamma': ('scale', 'auto'),
                                  'kernel': ('poly', 'rbf', 'linear')},
                      pre_dispatch='2*n_jobs', refit=True, return_train_score=False,
                      scoring=None, verbose=0)
```

```
In [53]: print(model)
```

```
GridSearchCV(cv=10, error_score='raise-deprecating',
             estimator=SVC(C=1.0, cache_size=200, class_weight=None, coef0=0.0,
                           decision_function_shape='ovr', degree=3,
                           gamma='auto_deprecated', kernel='rbf', max_iter=-1,
                           probability=False, random_state=None, shrinking=True,
                           tol=0.001, verbose=False),
             iid='warn', n_jobs=None,
             param_grid={'C': [0.1, 1, 10], 'gamma': ('scale', 'auto'),
                         'kernel': ('poly', 'rbf', 'linear')},
             pre_dispatch='2*n_jobs', refit=True, return_train_score=False,
             scoring=None, verbose=0)
```

```
I used GridSearch with CV=10 to evaluate this question. We see that the best model selected has C
 = 1, gamma = auto (which means gamma = 1 / number of features). Radial kerneling is chosen over p
olynomial kerneling (at degree =3) and linear kerneling.
```