Homework 6 - Aabir Abubaker Kar

March 8, 2020

0.1 Homework 6 - Aabir Abubaker Kar

Non-linear separation

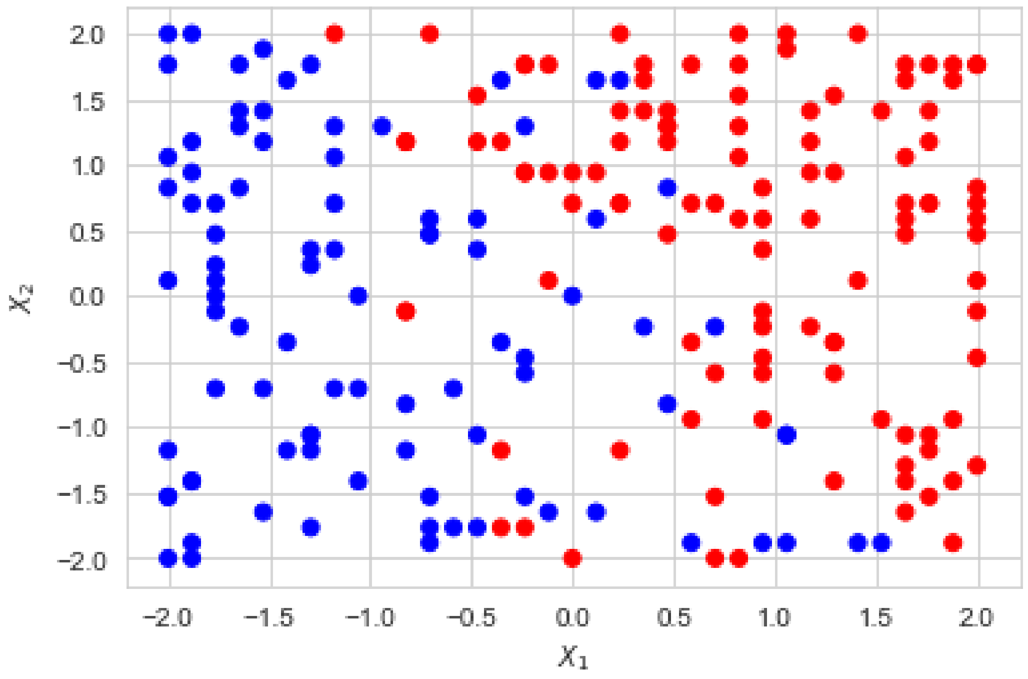
- (15 points) *Generate* a simulated two-class data set with 100 observations and two features in which there is a visible (clear) but still non-linear separation between the two classes. *Show* that in this setting, a support vector machine with a radial kernel will outperform a support vector classifier (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.

[184]:
```
import numpy as np
import matplotlib.pyplot as plt

from sklearn.svm import SVC
from sklearn.model_selection import train_test_split, cross_validate,␣
↪cross_val_score, GridSearchCV
from sklearn.metrics import accuracy_score, roc_auc_score
from sklearn.linear_model import LogisticRegressionCV
import pandas as pd

import tabulate as tb


import seaborn as sns

sns.set_style("whitegrid")

np.random.seed(42)

def plt_labels(x='', y='', title=''):
    plt.xlabel(x)
    plt.ylabel(y)
    plt.title(title)
```

[16]:
```
vals = np.linspace(-2, 2, 35)
x = []
for i in range(200):
    x.append([np.random.choice(vals), np.random.choice(vals)])
```

```
x = np.array(x)
y = (0.25*x[:, 1] + x[:, 0])+0.1*np.random.randn()>np.cos(30*x[:, 1])
```

[17]:
```
cols = ['red' if yval else 'blue' for yval in y]
plt.scatter(x[:, 0], x[:, 1], c=cols)
plt_labels(r'$X_1$', r'$X_2$')
plt.show()
```

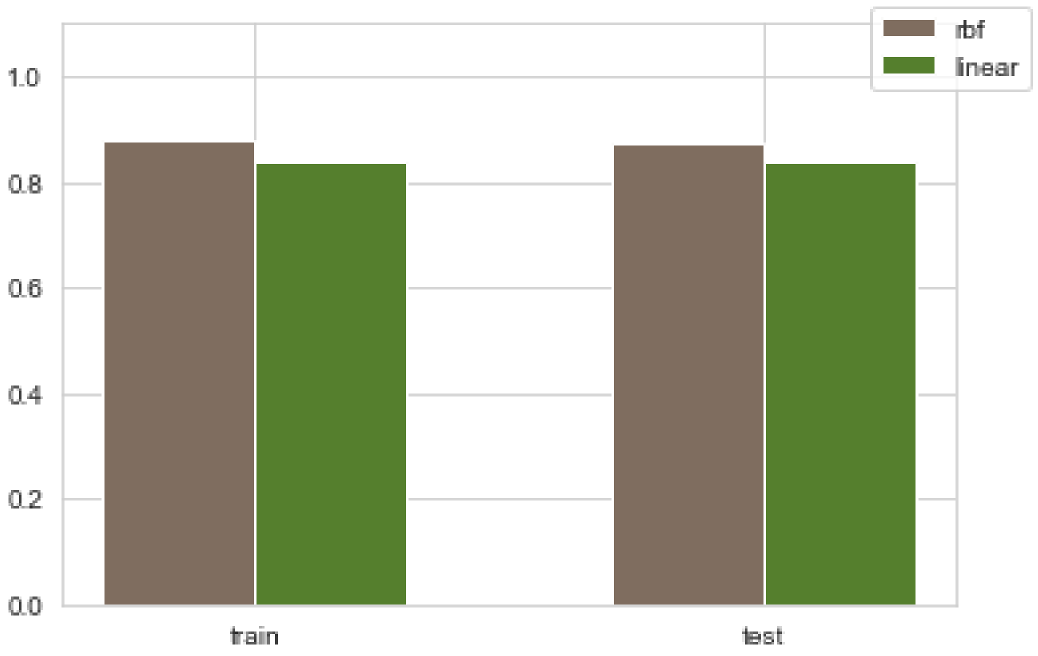


[22]:
```
def compare_kernels(x, y):
    x_train, x_test, y_train, y_test = train_test_split(x, y)
    print(f"train data shape: {x_train.shape}\ntest data shape: {x_test.shape}")
    models = []
    accs = {'test':[], 'train':[]}
    for kern in ['rbf', 'linear']:
        svc = SVC(kernel=kern).fit(x_train, y_train)
        models.append(svc)
        y_pred = svc.predict(x_test)
        accs['test'].append(accuracy_score(y_test, y_pred))
        accs['train'].append(accuracy_score(y_train, svc.predict(x_train)))
    return models, accs
```

[26]:
```
models, accs = compare_kernels(x, y)
```

```
print('\n')

print(tb.tabulate({'model':['rbf', 'linear'],
                   'train accuracy': accs['train'],
                   'test accuracy': accs['test']}, headers='keys'))

def pair_barplot(x1, x2, barWidth = 0.3):
    # Set position of bar on X axis
    r1 = np.arange(len(x1))
    r2 = [x + barWidth for x in r1]
    plt.bar(r1, x1, color='#7f6d5f', width=barWidth, edgecolor='white')
    plt.bar(r2, x2, color='#557f2d', width=barWidth, edgecolor='white')


bw=0.3
pair_barplot(accs['train'], accs['test'], bw)
plt.ylim(0, 1.1)
plt.xticks([r+bw/2 for r in range(2)], ['train', 'test'])
plt.legend(['rbf', 'linear'], bbox_to_anchor=(1.1, 1.05))
plt.show()
```

```
train data shape: (150, 2)
test data shape: (50, 2)


model      train accuracy    test accuracy
-------  ----------------  ---------------
rbf              0.88                 0.84
linear           0.873333             0.84
```



- The radial kernal clearly outperforms the linear one on the training data.
- Surprisingly, both have the same test accuracy. This can be explained by the fact that the test set is very small - it has only 50 observations and the variance of the test error is thus much higher

SVM vs. logistic regression We have seen that we can fit an SVM with a non-linear kernel in order to perform classification using a non-linear decision boundary. We will now see that we can also obtain a non-linear decision boundary by performing logistic regression using non-linear transformations of the features. Your goal here is to compare different approaches to estimating non-linear decision boundaries, and thus assess the benefits and drawbacks of each.

- ~~(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.~~
- ~~(5 points) Plot the observations with colors according to their class labels (y). Your plot should display X_1 on the x-axis and X_2 on the y-axis.~~
- ~~(5 points) Fit a logistic regression model to the data, using X_1 and X_2 as predictors.~~
- ~~(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*).~~
- ~~(5 points) Now fit a logistic regression model to the data, but this time using some *non-linear function* of both X_1 and X_2 as predictors (e.g. $X_1^2, X_1 \times X_2, \log(X_2)$, and so on).~~
- ~~(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~~

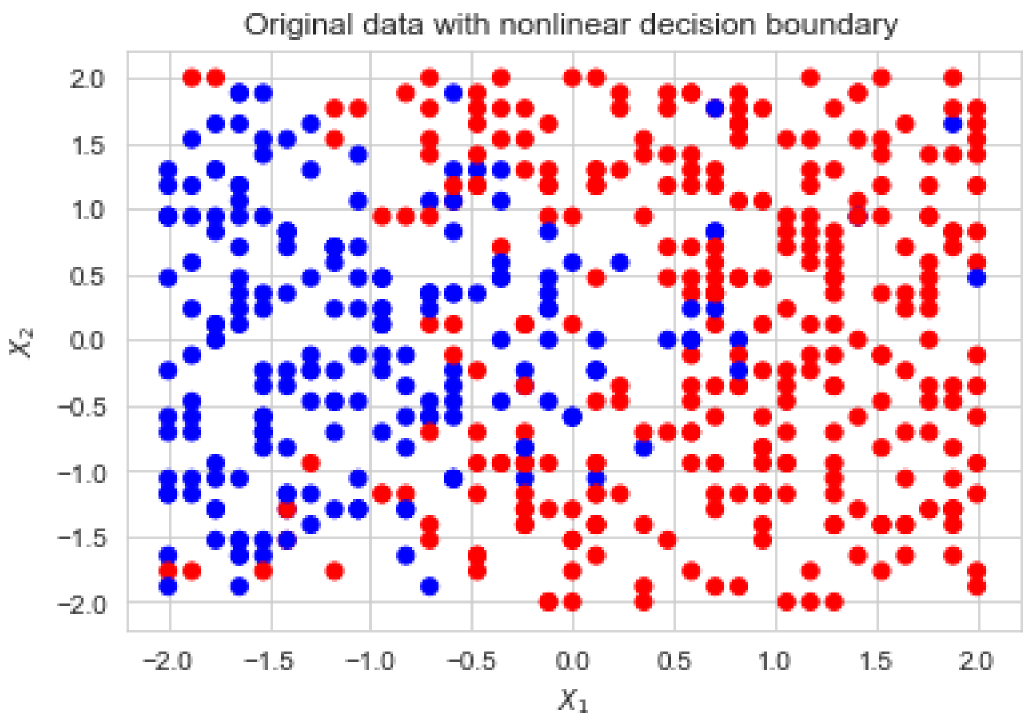
~~observations, colored according to the new *predicted* class labels from the non-linear model (*the decision boundary should now be obviously non-linear*). If it is not, then repeat earlier steps until you come up with an example in which the predicted class labels and the resultant decision boundary are clearly non-linear.~~

- (5 points) Now, fit a support vector classifier (linear kernel) to the data with *original* X_1 and X_2 as predictors. Obtain a class prediction for each observation. Plot the observations, colored according to the *predicted* class labels.
- (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.
- (5 points) Discuss your results and specifically the tradeoffs between estimating non-linear decision boundaries using these two different approaches.

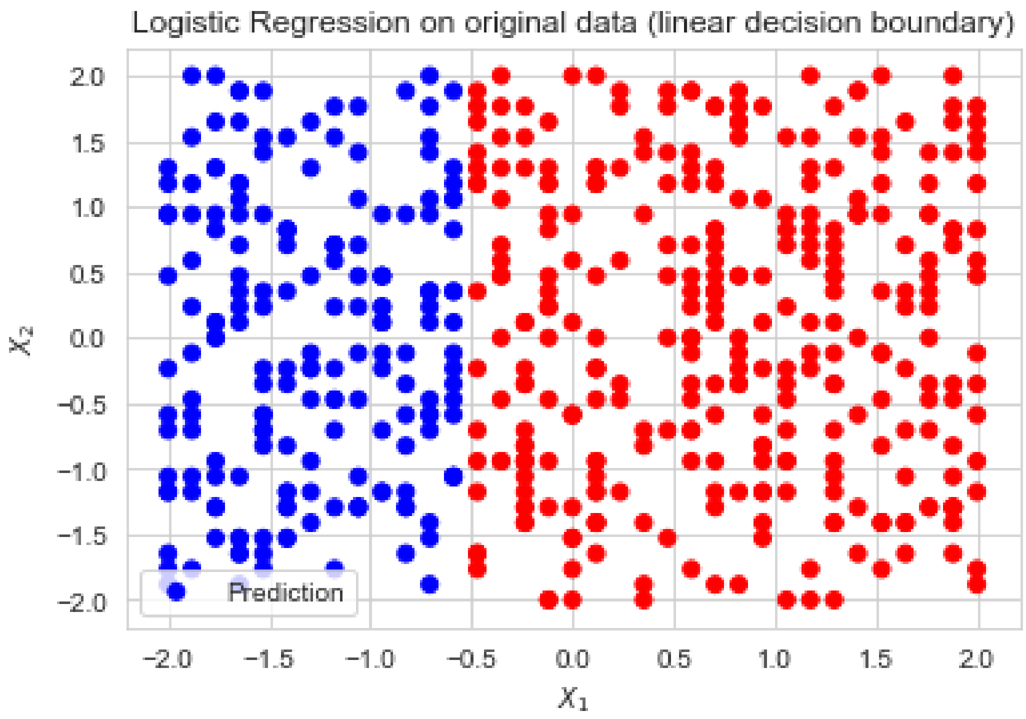
[86]:
```
vals = np.linspace(-2, 2, 35)
x = []
for i in range(500):
    x.append([np.random.choice(vals), np.random.choice(vals)])

x = np.array(x)

y = (0.5*np.power(x[:, 1],2) + x[:, 0])+0.4*np.random.randn()>np.cos(30*x[:, 1])

np.put(y, np.random.choice(range(len(y)), size=len(y)//40, replace=False), 1)
np.put(y, np.random.choice(range(len(y)), size=len(y)//40, replace=False), 0)

print(x.shape, y.shape)
```

```
(500, 2) (500,)
```

[105]:
```
cols = ['red' if yval else 'blue' for yval in y]
plt.scatter(x[:, 0], x[:, 1], c=cols)
plt_labels(r'$X_1$', r'$X_2$')
plt.title('Original data with nonlinear decision boundary')
plt.show()
```



[106]:
```
lr = LogisticRegressionCV(cv=5, random_state=0).fit(x, y.ravel())

y_pred = lr.predict(x)

cols = ['red' if yval else 'blue' for yval in y_pred]
plt.scatter(x[:, 0], x[:, 1], c=cols, label='Prediction')
plt_labels(r'$X_1$', r'$X_2$')
plt.legend()
plt.title('Logistic Regression on original data (linear decision boundary)')
plt.show()

print(tb.tabulate({'Logistic Regression accuracy': [lr.score(x, y)]},␣
 ↪headers='keys'))
```



```
  Logistic Regression accuracy
------------------------------
                         0.846
```

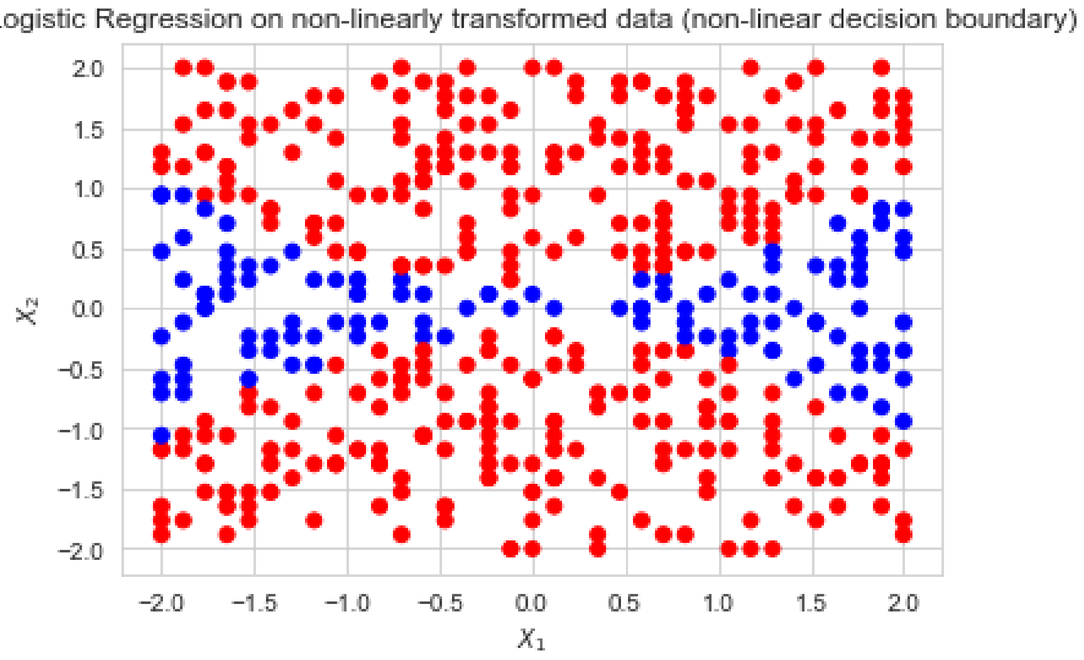
We can clearly see that logistic regression tries to find a linear decision boundary for the classification problem.

Adding non-linear features changes things:

```
[107]: x_nl = np.array([np.log(1+np.abs(x[:,1])),
                        np.power(x[:,0],2)
                                #+np.power(x[:,1],2)
                               ]).T

       print(x_nl.shape)
       #print(x_nl.min(), x_nl.max(), np.isnan(x_nl).any())

       lr = LogisticRegressionCV(cv=5, random_state=0).fit(x_nl, y.ravel())

       y_nl_pred = lr.predict(x_nl)

       cols = ['red' if yval else 'blue' for yval in y_nl_pred]
       plt.scatter(x[:, 0], x[:, 1], c=cols, label='Prediction')
       plt_labels(r'$X_1$', r'$X_2$')
```

```
plt.title('Logistic Regression on non-linearly transformed data (non-linear␣
↪decision boundary)')
#plt.legend()
plt.show()

print(tb.tabulate({'Logistic Regression accuracy with nonlinearity': [lr.
↪score(x_nl, y)]}, headers='keys'))
```

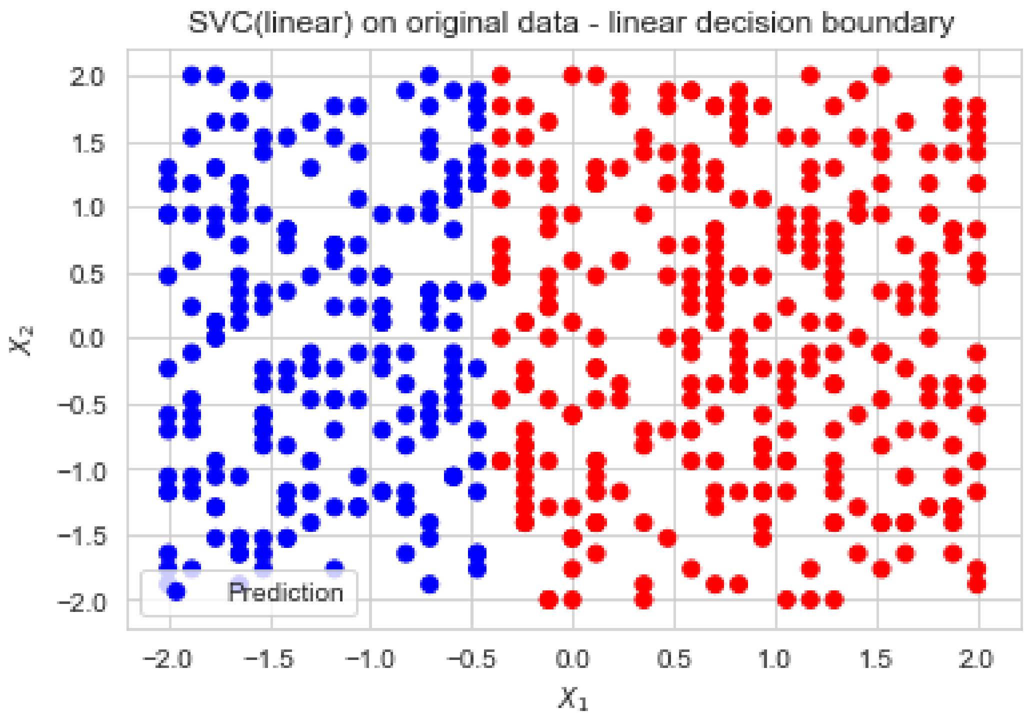
```
(500, 2)
```



```
  Logistic Regression accuracy with nonlinearity
------------------------------------------------
                                           0.618
```

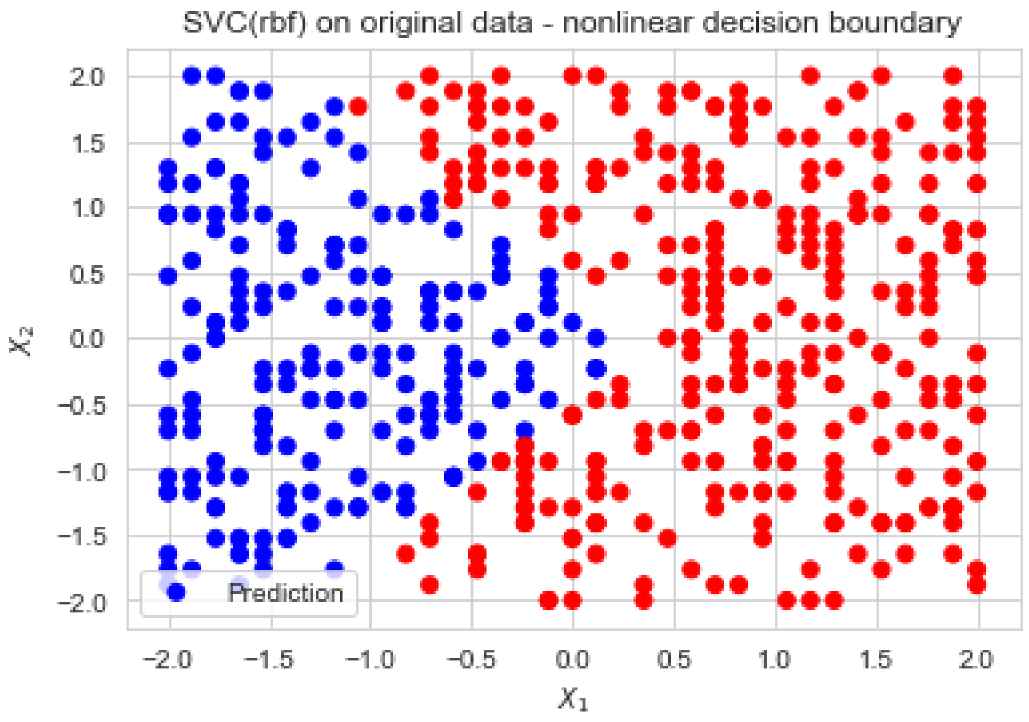
[112]:
```
svc = SVC(kernel='linear').fit(x, y)

y_pred = svc.predict(x)

cols = ['red' if yval else 'blue' for yval in y_pred]
plt.scatter(x[:, 0], x[:, 1], c=cols, label='Prediction')
plt_labels(r'$X_1$', r'$X_2$')
plt.legend()
plt.title('SVC(linear) on original data - linear decision boundary')
plt.show()
```

```
print(tb.tabulate({'Linear SVM accuracy':
                      [svc.score(x, y)]}, headers='keys'))
```



```
  Linear SVM accuracy
---------------------
                0.824
```

[113]:
```
svc = SVC(kernel='rbf').fit(x, y)

y_pred = svc.predict(x)

cols = ['red' if yval else 'blue' for yval in y_pred]
plt.scatter(x[:, 0], x[:, 1], c=cols, label='Prediction')
plt_labels(r'$X_1$', r'$X_2$')
plt.legend()
plt.title('SVC(rbf) on original data - nonlinear decision boundary')
plt.show()

print(tb.tabulate({'Nonlinear SVM accuracy':
                      [svc.score(x, y)]}, headers='keys'))
```



```
  Nonlinear SVM accuracy
------------------------
                   0.874
```

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

Clearly the non-linear kernel on SVM works best on data with a nonlinear decision boundary.

This is made evident by the fact that linear kernels can only identify linear decision boundaries. Applying non-linear transformations to the data before using linear classifiers is a potential work-around. However, we see that the features could not well-represent the non-linearities truly causing the data. This explains the sharp drop in accuracy from 85% using Logistic regression on raw data, to 62% using it on non-linearly transformed data.

Intuitively, the use of squaring and logarithms don't reflect the original cosine dependence in the data generating process.

When the decision boundary is non-linear, a non-linear SVM does a great job of generalizing the learning. It is only if we have well-motivated (theoretically or otherwise) functional forms of specific non-linear features that it makes sense to use these on a linear model instead.

Tuning cost

In class we learned that in the case of data that is just barely linearly separable, a support vector classifier with a small value of cost that misclassifies a couple of training observations may perform better on test data than one with a huge value of cost that does not misclassify any training observations. You will now investigate that claim.

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

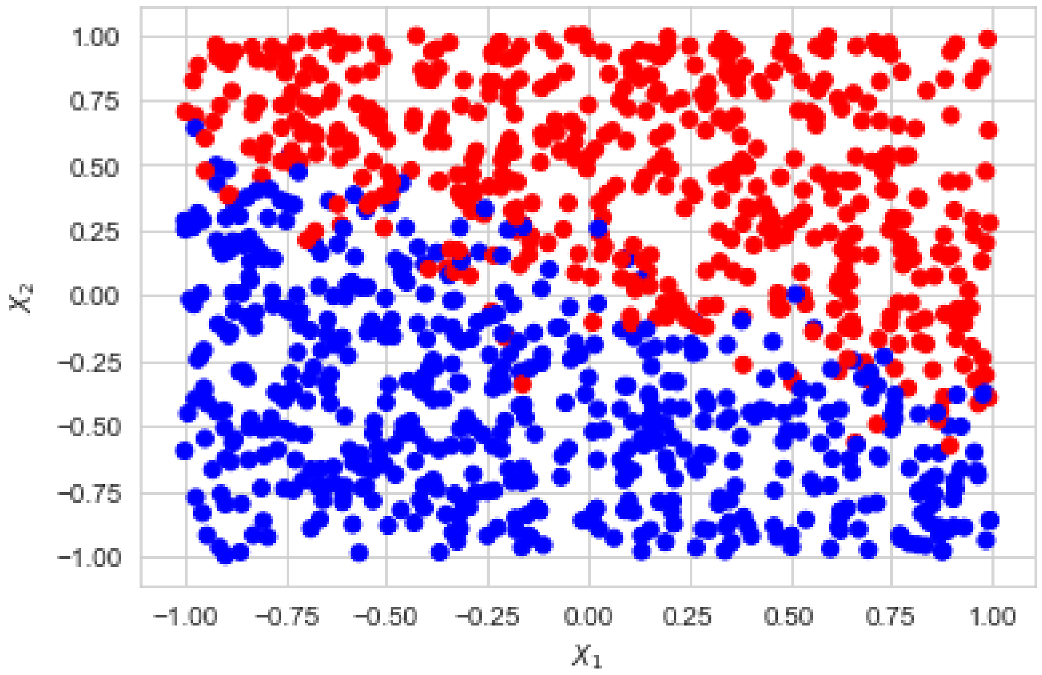
(5 points) Compute the cross-validation error rates for support vector classifiers with a range of cost values. How many training errors are made for each value of cost considered, and how does this relate to the cross-validation errors obtained?

(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?

(5 points) Discuss your results.

[143]:
```
x = np.array([np.random.uniform(-1, 1, 1000), np.random.uniform(-1, 1, 1000)]).T

y = x[:,0]+2*x[:,1]+ 0.3*np.random.randn(1000)
y = (np.exp(y)/(1+np.exp(y)))>0.5

print(x.shape, y.shape)

x_train, x_test, y_train, y_test = train_test_split(x, y)

cols = ['red' if yval else 'blue' for yval in y]
plt.scatter(x[:, 0], x[:, 1], c=cols)
plt_labels(r'$X_1$', r'$X_2$')
plt.show()
```

```
(1000, 2) (1000,)
```



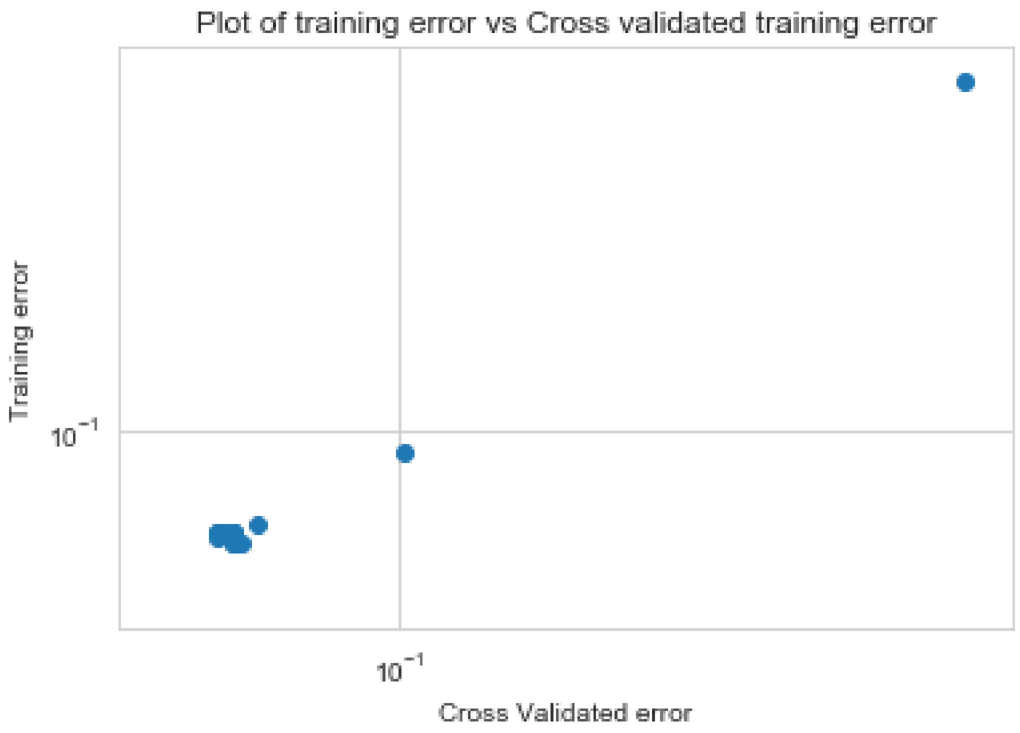
```
[ ]: def cost_cross_val(x, y):
         training_errors=[]
         models = []
         CV_errors = []
         Cs = list(np.logspace(-3, -0.001, 5))+list(range(1, 11))
         for c in Cs:
             model = SVC(kernel='linear', C = c, random_state=0)
             err = 1-np.mean(cross_val_score(model, x, y, cv=10, scoring='accuracy'))
             models+= [model]
             CV_errors+= [err]
             training_errors+= [1-model.fit(x,y).score(x,y)]
         minarg = np.argmin(CV_errors)
         min_error = CV_errors[minarg]
         best_model = models[minarg]
         best_C = Cs[minarg]
         return best_model, best_C, min_error, Cs, training_errors, CV_errors

     best_model, best_C, min_CV_error, Cs, training_errors, CV_errors =␣
      ↪cost_cross_val(x_train, y_train)
```

```
[165]: print(tb.tabulate({'cost':Cs, 'training error':training_errors,
                         'Cross validated error': CV_errors
                         #, 'CS error': CV_errors
                        }, headers='keys'))
```

```
       cost    training error    Cross validated error
-----------  ----------------  -----------------------
 0.001              0.462667                 0.462667
 0.00562018         0.0906667                0.101333
 0.0315864          0.0666667                0.068
 0.177521           0.064                    0.064
 0.9977             0.064                    0.0613333
 1                  0.064                    0.0613333
 2                  0.064                    0.0613333
 3                  0.0626667                0.0613333
 4                  0.064                    0.064
 5                  0.064                    0.0626667
 6                  0.064                    0.064
 7                  0.0613333                0.0653333
 8                  0.0613333                0.064
 9                  0.0613333                0.064
10                  0.0613333                0.064
```

[166]:
```python
plt.scatter(CV_errors, training_errors)
plt.ylabel('Training error')
plt.xlabel('Cross Validated error')
plt.xscale('log')
plt.yscale('log')
plt.title('Plot of training error vs Cross validated training error')
```

[166]:
```
Text(0.5, 1.0, 'Plot of training error vs Cross validated training error')
```

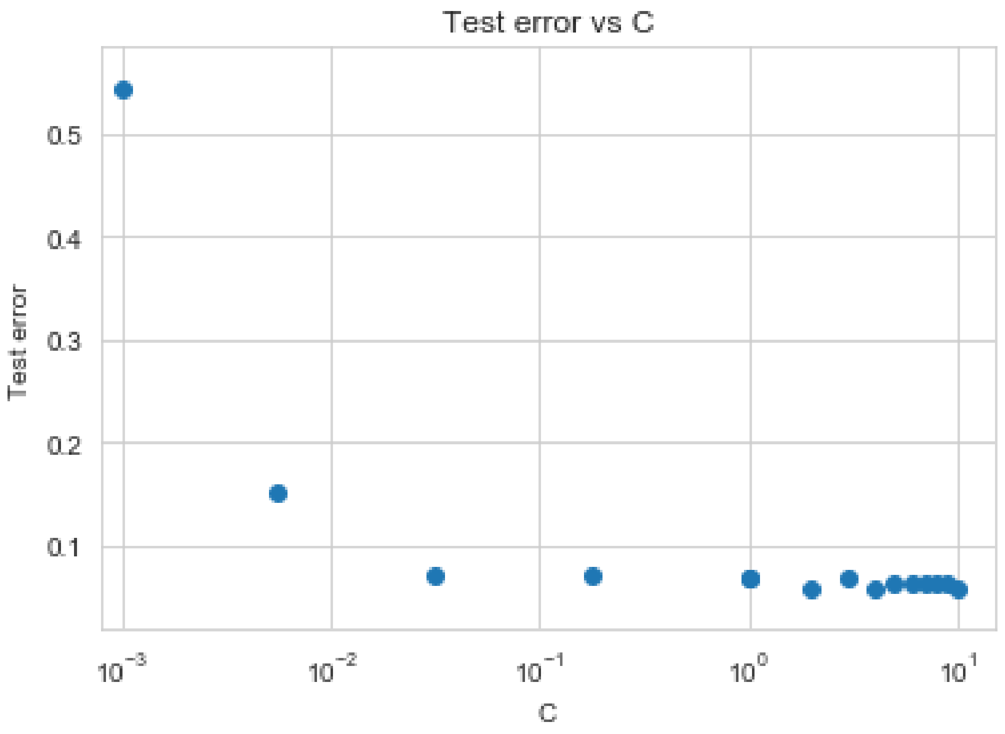


[167]:
```
print("The cross-validated error is lowest at C=", best_C)
print("The training error is lowest at C=", Cs[np.argmin(CV_errors)])
```

```
The cross-validated error is lowest at C= 0.9977000638225533
The training error is lowest at C= 0.9977000638225533
```

The above graph shows that there is a largely linear relationship between training errors and cross-validated errors for each value of C. In fact, the values are almost equal. This is consistent with our expectation that the data are already linearly separable, and therefore we expect both methods to perform equally well. I feel like one might be able to avoid using cross-validation as a way to 'fine-tune' methods that are working fine.

[168]:
```
svm_testerr = []
for c in Cs:
    model = SVC(C=c, random_state=0).fit(x_train, y_train)
    error = 1 - model.score(x_test, y_test)
    svm_testerr.append(error)

plt.clf()
plt.scatter(Cs, svm_testerr)
plt.ylabel('Test error')
plt.xlabel('C')
plt.xlim(0.8*min(Cs), 1.5*max(Cs))
plt.xscale('log')
plt.title('Test error vs C')
plt.show()
```



[169]:
```
print("Lowest test error for C ="
      , Cs[np.argmin(svm_testerr)])
```

```
Lowest test error for C = 2
```

These results tell us that the cost is a less influential variable on our algorithm when the underlying assumption of linearity is well-motivated (here, we literally generate the data ourself).

Due to this simple validation, incorrect predictions don't have to be punished too strongly, and a low value of C (over 0.01) seems to be good enough to achieve a significantly low test error.

0.2 Applications

(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.

[170]:
```
dftrain = pd.read_csv('./data/gss_train.csv')
dftest = pd.read_csv('./data/gss_test.csv')

x_tr, x_te = dftrain.drop(columns="colrac"), dftest.drop(columns="colrac")
y_tr, y_te = dftrain["colrac"], dftest["colrac"]
```

[176]:
```
best_model, best_C, min_CV_error, Cs, training_errors, CV_errors =␣
 ↪cost_cross_val(x_tr, y_tr)
```

```
Best tuning parameter: 0.03158639048423472 , with CV error =  0.1991882822419735

 Training Errors: [0.22687373396353816, 0.1971640783254558, 0.18568534773801482,
0.1782579338284943, 0.18095881161377447, 0.18095881161377447,
0.18095881161377447, 0.18163403106009457, 0.18095881161377447,
0.18163403106009457, 0.18095881161377447, 0.18095881161377447,
0.18095881161377447, 0.18095881161377447, 0.18163403106009457]

 CV Errors: [0.23970161436604376, 0.20730092508615994, 0.1991882822419735,
0.2039089424995466, 0.20593143479049525, 0.20593143479049525,
0.20660711046617086, 0.20593143479049525, 0.2079584618175223,
0.20863413749319792, 0.20863413749319792, 0.20863413749319792,
0.20930981316887354, 0.20930981316887354, 0.20930981316887354]
```

[181]:
```python
print("Best tuning parameter:", best_C, ", with CV error = ",␣
 ↪min_CV_error,"\n"*2)

print(tb.tabulate({'Cost': Cs, "training error": training_errors, "CV error":␣
 ↪CV_errors}
                 , headers='keys'))
```

```
Best tuning parameter: 0.03158639048423472 , with CV error =  0.1991882822419735


      Cost    training error    CV error
-----------  ----------------  ----------
 0.001               0.226874    0.239702
 0.00562018          0.197164    0.207301
 0.0315864           0.185685    0.199188
 0.177521            0.178258    0.203909
 0.9977              0.180959    0.205931
 1                   0.180959    0.205931
 2                   0.180959    0.206607
 3                   0.181634    0.205931
 4                   0.180959    0.207958
 5                   0.181634    0.208634
 6                   0.180959    0.208634
 7                   0.180959    0.208634
 8                   0.180959    0.20931
 9                   0.180959    0.20931
10                   0.181634    0.20931
```

[182]:
```python
print("CV errors for values of cost")
print(tb.tabulate({'cost':Cs, 'CV error':CV_errors}
                 , headers='keys'))
```

```
CV errors for values of cost
      cost    CV error
```

```
-----------  ----------
 0.001         0.239702
 0.00562018    0.207301
 0.0315864     0.199188
 0.177521      0.203909
 0.9977        0.205931
 1             0.205931
 2             0.206607
 3             0.205931
 4             0.207958
 5             0.208634
 6             0.208634
 7             0.208634
 8             0.20931
 9             0.20931
10             0.20931
```

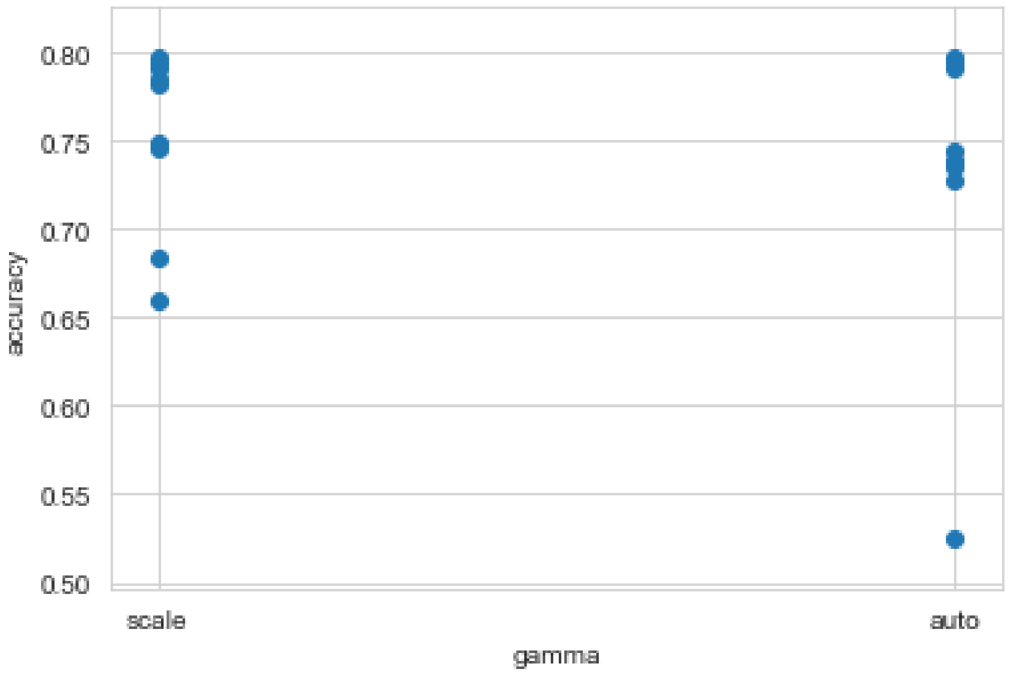
(15 points) Repeat the previous question, but this time using SVMs with radial and polynomial basis kernels, with different values for gamma and degree and cost. Present and discuss your results (e.g., fit, compare kernels, cost, substantive conclusions across fits, etc.).

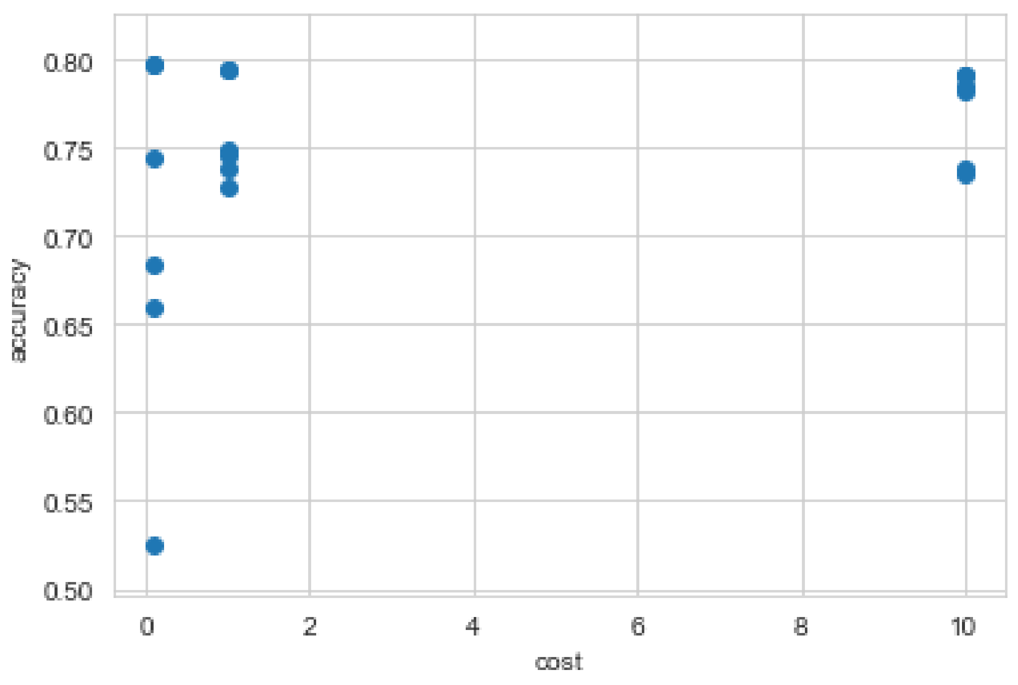
[186]:
```
parameters = {'kernel':('poly', 'rbf', 'linear'), 'C':[0.1, 1, 10], 'gamma':
 ↪('scale','auto')}

model = GridSearchCV(SVC(), parameters, cv=10, verbose=1)

model.fit(x_tr, y_tr)

print(model)
```

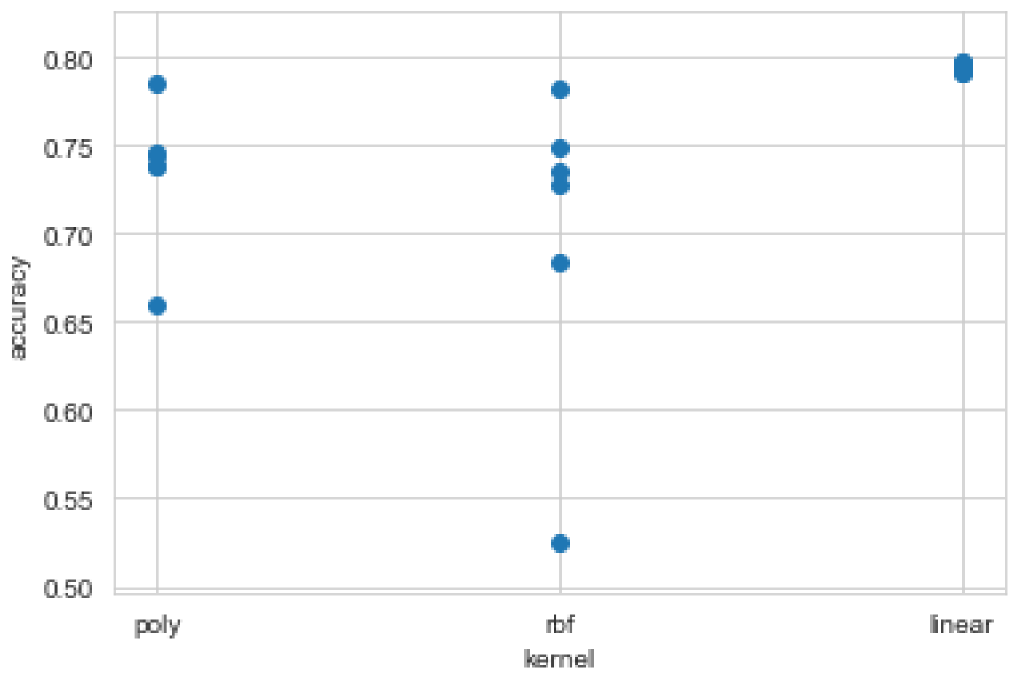
```
Fitting 10 folds for each of 18 candidates, totalling 180 fits

[Parallel(n_jobs=1)]: Using backend SequentialBackend with 1 concurrent workers.
[Parallel(n_jobs=1)]: Done 180 out of 180 | elapsed:  8.8min finished

GridSearchCV(cv=10, error_score=nan,
             estimator=SVC(C=1.0, break_ties=False, cache_size=200,
                           class_weight=None, coef0=0.0,
                           decision_function_shape='ovr', degree=3,
                           gamma='scale', kernel='rbf', max_iter=-1,
                           probability=False, random_state=None, shrinking=True,
                           tol=0.001, verbose=False),
             iid='deprecated', 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=1)
```

[192]:
```
results = model.cv_results_
#print(results.keys())
scores = results['mean_test_score']
gammas, Cs, kernels = results['param_gamma'], results['param_C'],␣
 ↪results['param_kernel']
```

[194]:
```
plt.scatter(gammas, scores)
plt_labels('gamma', 'accuracy')
plt.show()
plt.scatter(Cs, scores)
plt_labels('cost', 'accuracy')
plt.show()
plt.scatter(kernels, scores)
plt_labels('kernel', 'accuracy')
plt.show()
```







We can interpret these graphs as telling us that the variance on a particular parameter being low (all other things being equal), means that that value of the parameter is the ideal choice.

Therefore, for the gamma parameter, we find ‘scale’ to be the better option. Both methods provide high accuracy but ‘scale’ has lower variance, compared with the outlying low accuracy provided by once instance of ‘auto’.

Similarly, for cost, we prefer larger values (between 1 and 10), as they are more robust and show less variance in accuracy.

For the kernal choice, we would prefer a linear one in this case (as we know the data are linearly separable and the algorithm validates this). For cases that cannot be justified to be linear, the rbf or poly options should still be tested.

```
[ ]:
```