In [55]:

```
# Imports

import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import GridSearchCV
from sklearn.pipeline import make_pipeline
from sklearn.decomposition import KernelPCA
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import classification_report
from sklearn.metrics import confusion_matrix
from sklearn.model_selection import cross_val_predict
from sklearn.model_selection import StratifiedKFold
from sklearn.metrics import roc_curve, auc
from sklearn.metrics import roc_auc_score
from sklearn.svm import SVC
from sklearn.model_selection import learning_curve
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import validation_curve
```

Tuning and evaluating Glass composition

Explorative data analysis and preprocessing

In [2]:

```
df = pd.read_csv('CA4-glassTrain.csv')
dfTest = pd.read_csv('CA4-glassTest.csv')
```

In [3]:

```
df.head()
```

Out[3]:

	Unnamed: 0	RI	Na	Mg	Al	Si	K	Ca	Ba	Fe	type
0	0	1.52101	13.64	4.49	1.10	71.78	0.06	8.75	0.0	0.0	1
1	1	1.51761	13.89	3.60	1.36	72.73	0.48	7.83	0.0	0.0	1
2	2	1.51766	13.21	3.69	1.29	72.61	0.57	8.22	0.0	0.0	1
3	3	1.51742	13.27	3.62	1.24	73.08	0.55	8.07	0.0	0.0	1
4	4	1.51743	13.30	3.60	1.14	73.09	0.58	8.17	0.0	0.0	1

In [4]:

```
# Search for missing values
missing = np.asarray(df.isnull().sum())
if missing.any():
    print("Dataset has missing values")
else:
    print('No missing values!')
```

```
No missing values!
```

In [22]:

```
# Number of features to be plotted

num_vars = 9

# Create frame with features and types
glass_df = df.iloc[:, 1:num_vars+2]
# assign types to variable y
y = glass_df.iloc[:, 9]
# assign feature to variable X
X = glass_df.iloc[:, 0:num_vars]
```

In [6]:

```
glass_df
```

Out[6]:

	RI	Na	Mg	Al	Si	K	Ca	Ba	Fe	type
0	1.52101	13.64	4.49	1.10	71.78	0.06	8.75	0.00	0.0	1
1	1.51761	13.89	3.60	1.36	72.73	0.48	7.83	0.00	0.0	1
2	1.51766	13.21	3.69	1.29	72.61	0.57	8.22	0.00	0.0	1
3	1.51742	13.27	3.62	1.24	73.08	0.55	8.07	0.00	0.0	1
4	1.51743	13.30	3.60	1.14	73.09	0.58	8.17	0.00	0.0	1
...	...	...	...	...	...	...	...	...	...	...
138	1.51831	14.39	0.00	1.82	72.86	1.41	6.47	2.88	0.0	7
139	1.51640	14.37	0.00	2.74	72.85	0.00	9.45	0.54	0.0	7
140	1.51685	14.92	0.00	1.99	73.06	0.00	8.40	1.59	0.0	7
141	1.52065	14.36	0.00	2.02	73.42	0.00	8.44	1.64	0.0	7
142	1.51711	14.23	0.00	2.08	73.36	0.00	8.62	1.67	0.0	7

143 rows × 10 columns

Get the unique glass types

In [7]:

```
glass_type = glass_df['type']
np.unique(glass_type)
```

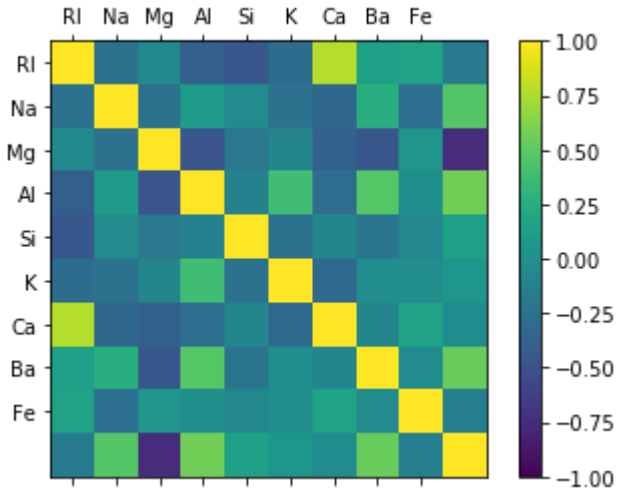
Out[7]:

```
array([1, 2, 3, 5, 6, 7], dtype=int64)
```

We see that there is no glass of type 4

In [8]:

```
# Plot correlation matrix

correlations = glass_df.corr()

fig = plt.figure()
ax = fig.add_subplot(111)
cax = ax.matshow(correlations, vmin=-1, vmax=1)
fig.colorbar(cax)
ticks = np.arange(0, num_vars, 1)
ax.set_xticks(ticks)
ax.set_yticks(ticks)
ax.set_xticklabels(list(glass_df.columns))
ax.set_yticklabels(list(glass_df.columns))
plt.show()
```

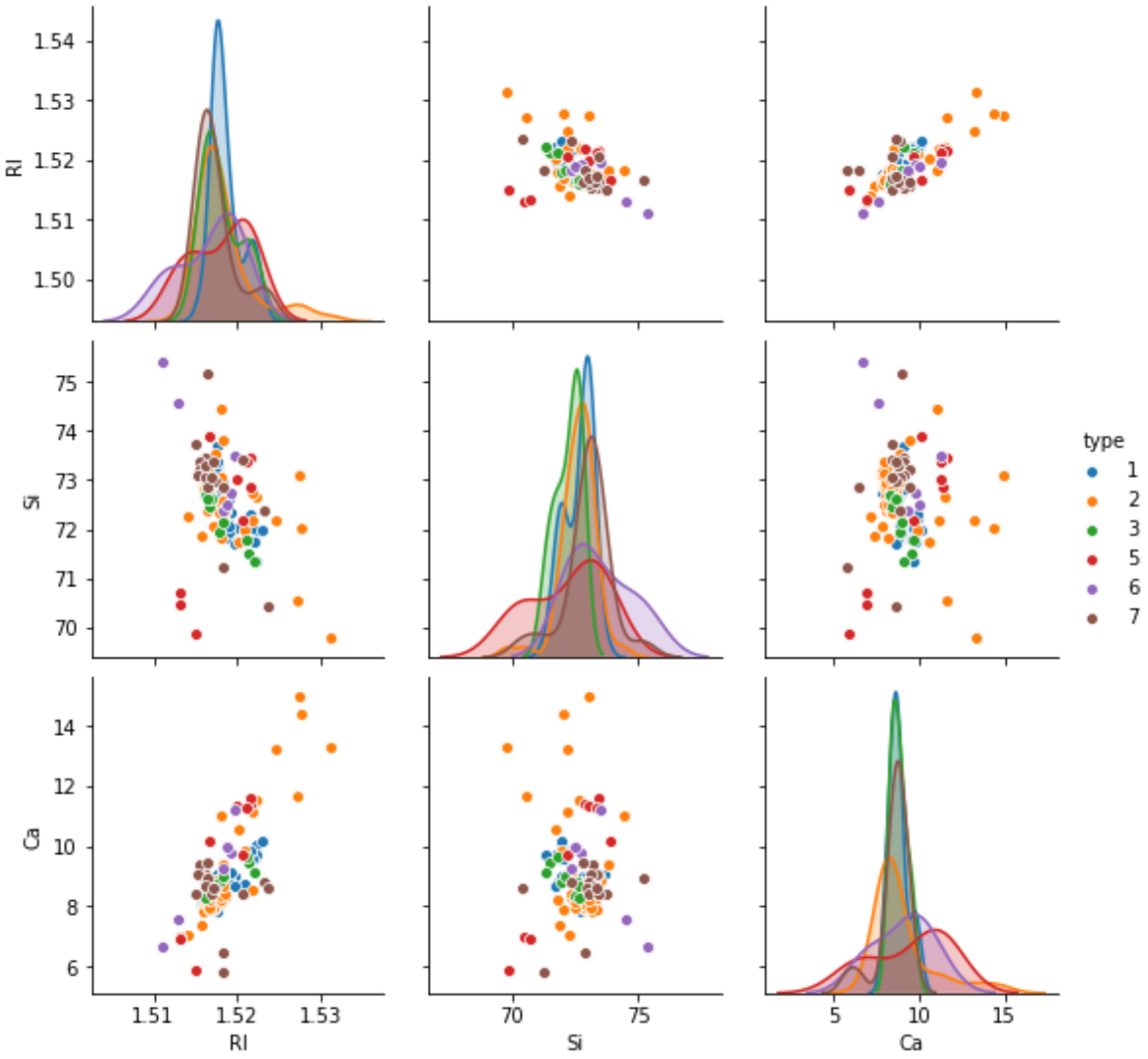


In [9]:

```
# Plotting for some features. Make new df
glass_df2 = glass_df.iloc[:, [0, 4, 6, 9]]
```

In [10]:

```
# Pairplot with seaborn
sns.pairplot(glass_df2, hue='type')
plt.show()
```



We can see there is a positive correlation between Ca and Ri in the upper right or lower left plot

In [11]:

```
# Group by type
group_type = glass_df['type'].groupby(glass_df['type'])
group_type.count()
```

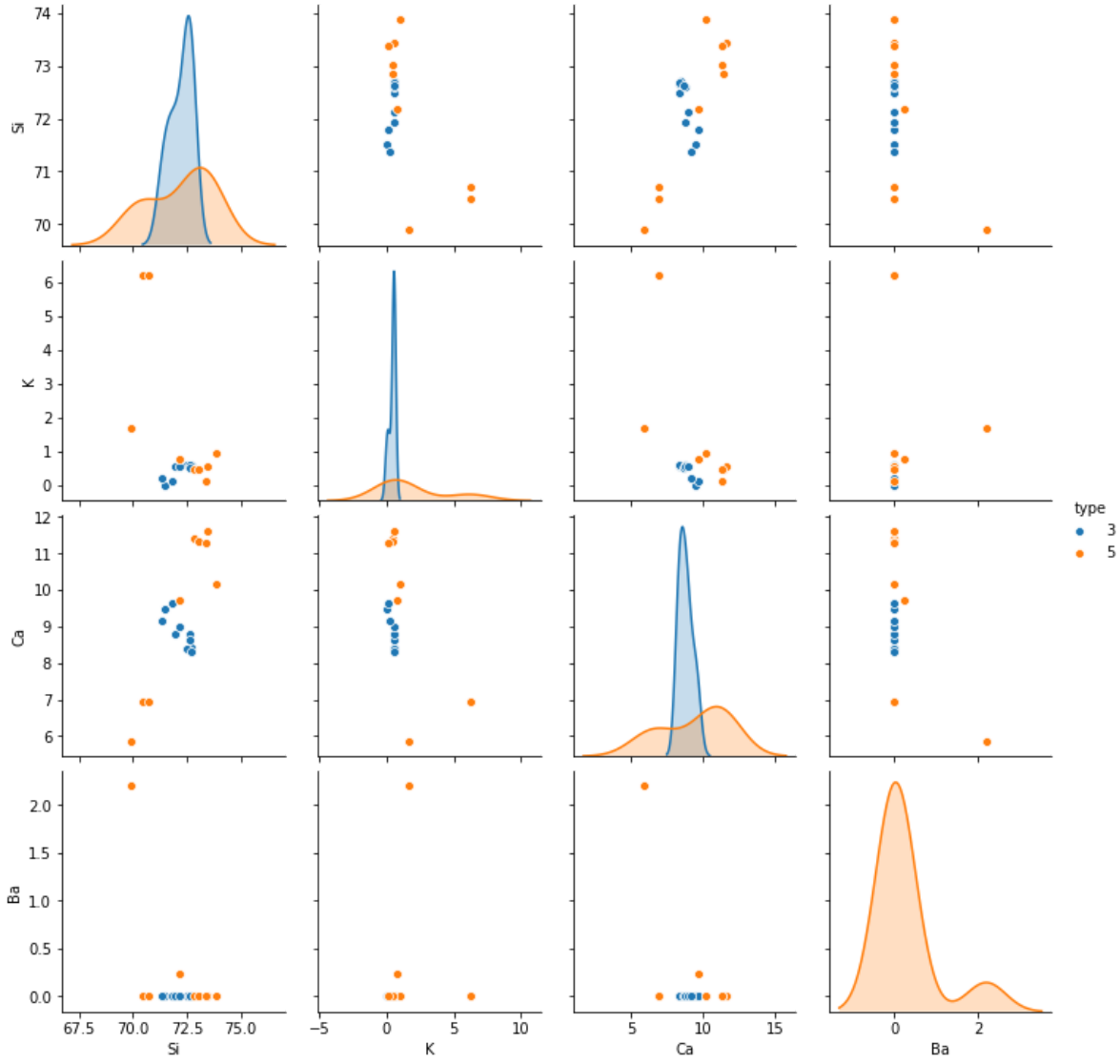
Out[11]:

```
type
1    47
2    51
3    11
5     9
6     6
7    19
Name: type, dtype: int64
```

In [12]:

```
# make plots for type 3 and 5 some of the features
glass_df35 = glass_df.iloc[:, [4, 5, 6, 7, 9]][98:118]
sns.pairplot(glass_df35, hue='type')
plt.show()
```

```
C:\Users\ander\miniconda3\lib\site-packages\seaborn\distributions.py:283: Us
erWarning: Data must have variance to compute a kernel density estimate.
  warnings.warn(msg, UserWarning)
```



Possible outliers for Ba and K

In [13]:

```
1 glass_df[98:118]
```

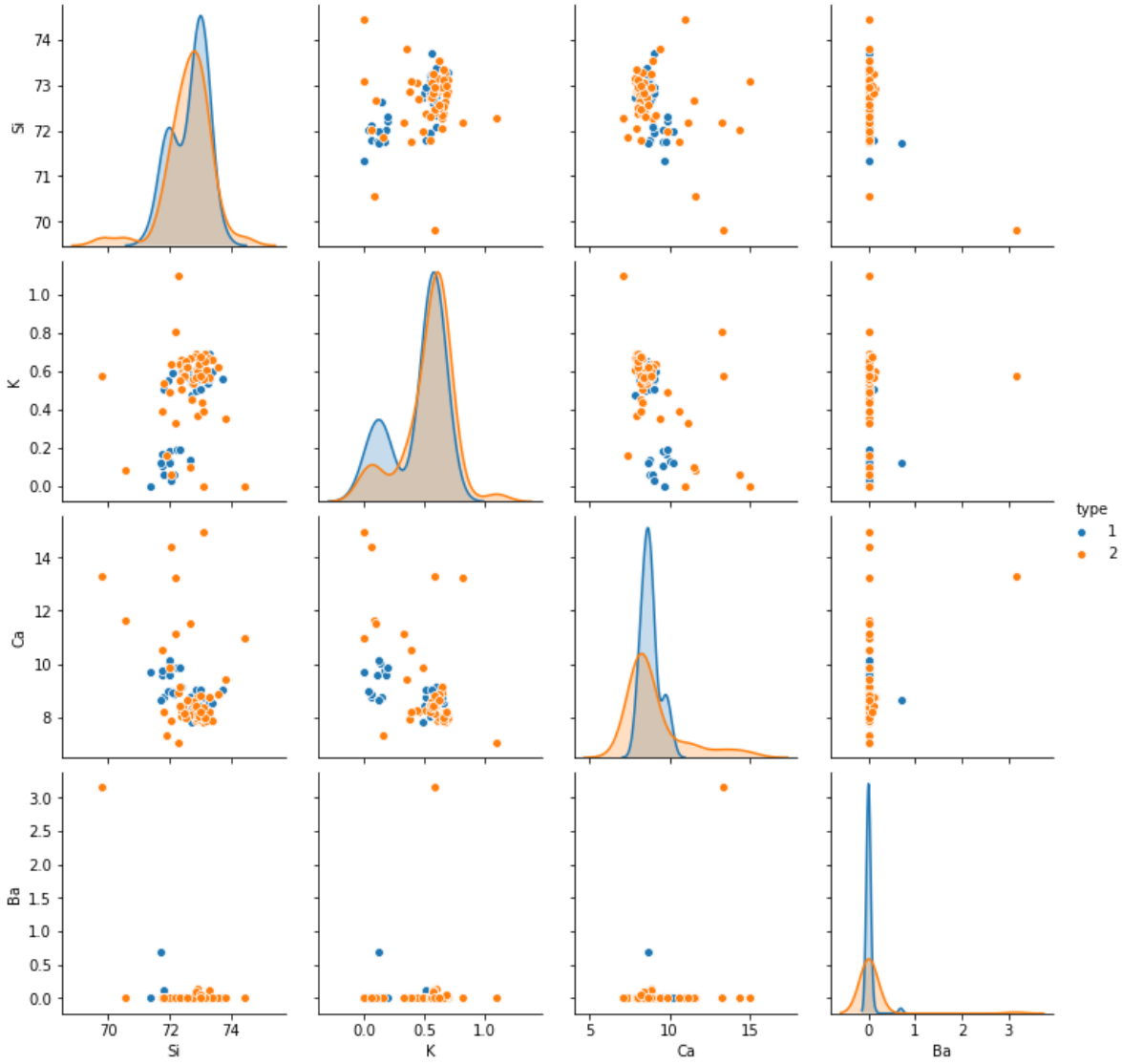
Out[13]:

	RI	Na	Mg	Al	Si	K	Ca	Ba	Fe	type
98	1.51610	13.33	3.53	1.34	72.67	0.56	8.33	0.00	0.00	3
99	1.51670	13.24	3.57	1.38	72.70	0.56	8.44	0.00	0.10	3
100	1.51665	13.14	3.45	1.76	72.48	0.60	8.38	0.00	0.17	3
101	1.52127	14.32	3.90	0.83	71.50	0.00	9.49	0.00	0.00	3
102	1.51610	13.42	3.40	1.22	72.69	0.59	8.32	0.00	0.00	3
103	1.51694	12.86	3.58	1.31	72.61	0.61	8.79	0.00	0.00	3
104	1.51655	13.41	3.39	1.28	72.64	0.52	8.65	0.00	0.00	3
105	1.52121	14.03	3.76	0.58	71.79	0.11	9.65	0.00	0.00	3
106	1.51796	13.50	3.36	1.63	71.94	0.57	8.81	0.00	0.09	3
107	1.51832	13.33	3.34	1.54	72.14	0.56	8.99	0.00	0.00	3
108	1.52211	14.19	3.78	0.91	71.36	0.23	9.14	0.00	0.37	3
109	1.51514	14.01	2.68	3.50	69.89	1.68	5.87	2.20	0.00	5
110	1.52171	11.56	1.88	1.56	72.86	0.47	11.41	0.00	0.00	5
111	1.52151	11.03	1.71	1.56	73.44	0.58	11.62	0.00	0.00	5
112	1.51666	12.86	0.00	1.83	73.88	0.97	10.17	0.00	0.00	5
113	1.51994	13.27	0.00	1.76	73.03	0.47	11.32	0.00	0.00	5
114	1.51316	13.02	0.00	3.04	70.48	6.21	6.96	0.00	0.00	5
115	1.51321	13.00	0.00	3.02	70.70	6.21	6.93	0.00	0.00	5
116	1.52058	12.85	1.61	2.17	72.18	0.76	9.70	0.24	0.51	5
117	1.52119	12.97	0.33	1.51	73.39	0.13	11.27	0.00	0.28	5

Can see that example 109 have relatively high Ba compared to rest of the class.
And 114 and 115 have high K compared to rest of the class

In [14]:

```
# make plots for type 1 and 2 some of the features
glass_df12 = glass_df.iloc[:, [4, 5, 6, 7, 9]][0:98]
sns.pairplot(glass_df12, hue='type')
plt.show()
```



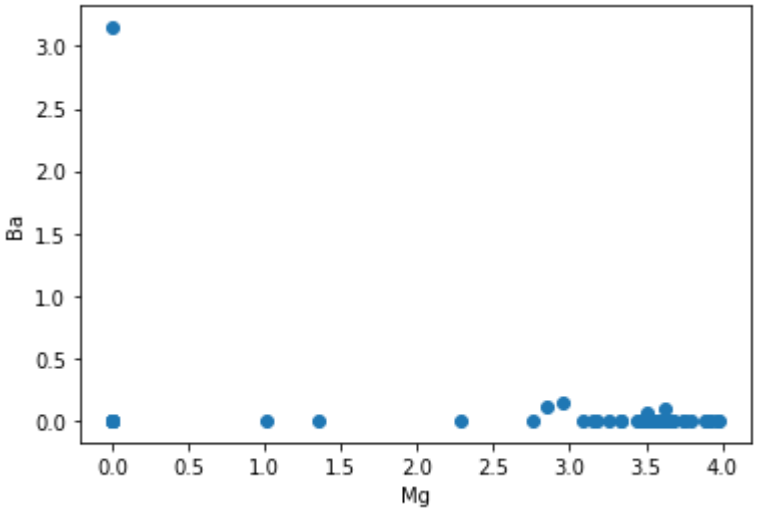
In the last line it seems to be outliers for glass type 1 and 2.
The majority of examples for both types contain nothing or only minimal Ba.

In [15]:

```
# Visualize Ba, mg in type 2
glass_df[['Mg', 'Ba', 'type']][47:98]
plt.scatter(glass_df['Mg'][47:98], glass_df['Ba'][47:98])
plt.xlabel('Mg')
plt.ylabel('Ba')
```

Out[15]:

```
Text(0, 0.5, 'Ba')
```



Another plot illustrating outlier in type 2. Plot of Mg and Ba. The outlier is positiond in the upper left while most of the data is contained in the lower right.

In [16]:

```
print('type 3:', glass_df['Ba'][98:109].describe())
print()
print('type 5:',glass_df['Ba'][109:118].describe())
```

```
type 3: count    11.0
mean      0.0
std       0.0
min       0.0
25%       0.0
50%       0.0
75%       0.0
max       0.0
Name: Ba, dtype: float64

type 5: count    9.000000
mean     0.271111
std      0.727675
min      0.000000
25%      0.000000
50%      0.000000
75%      0.000000
max      2.200000
Name: Ba, dtype: float64
```

In [17]:

```
# Look at data for 109 compare
glass_df[109:112]
```

Out[17]:

	RI	Na	Mg	Al	Si	K	Ca	Ba	Fe	type
109	1.51514	14.01	2.68	3.50	69.89	1.68	5.87	2.2	0.0	5
110	1.52171	11.56	1.88	1.56	72.86	0.47	11.41	0.0	0.0	5
111	1.52151	11.03	1.71	1.56	73.44	0.58	11.62	0.0	0.0	5

Ba in 109 is between two and three standard deviations away from the mean

In [18]:

```
# Describe all data of type 5
glass_df[109:118].describe()
```

Out[18]:

	RI	Na	Mg	Al	Si	K	Ca	Ba	
count	9.000000	9.000000	9.000000	9.000000	9.000000	9.000000	9.000000	9.000000	9.00
mean	1.518122	12.730000	0.912222	2.216667	72.205556	1.942222	9.472222	0.271111	0.08
std	0.003586	0.896856	1.051936	0.765849	1.476703	2.457493	2.272200	0.727675	0.18
min	1.513160	11.030000	0.000000	1.510000	69.890000	0.130000	5.870000	0.000000	0.00
25%	1.515140	12.850000	0.000000	1.560000	70.700000	0.470000	6.960000	0.000000	0.00
50%	1.519940	12.970000	0.330000	1.830000	72.860000	0.760000	10.170000	0.000000	0.00
75%	1.521190	13.020000	1.710000	3.020000	73.390000	1.680000	11.320000	0.000000	0.00
max	1.521710	14.010000	2.680000	3.500000	73.880000	6.210000	11.620000	2.200000	0.51

In [23]:

```
# Continue looking at possible outliers for Ba in type 1 and 2
print('type 1:', glass_df['Ba'][:47].describe())
print()
print('type 2:', glass_df['Ba'][47:98].describe())
```

```
type 1: count    47.000000
mean      0.017021
std       0.101573
min       0.000000
25%       0.000000
50%       0.000000
75%       0.000000
max       0.690000
Name: Ba, dtype: float64

type 2: count    51.000000
mean      0.069608
std       0.440881
min       0.000000
25%       0.000000
50%       0.000000
75%       0.000000
max       3.150000
Name: Ba, dtype: float64
```

The max value of Ba in glass type 1 is more than six standard deviations greater than its mean. We see that the same is true for type 2

We try removing some of the examples containing outliers for the respective types. We focus on examples with an extraordinary high value of Ba.

In [24]:

```
# Find index of max value of Ba in type 1
# This index will be the same as index for the entire dataset
Ba_series_type1 = glass_df['Ba'][:47]
Ba_series_type1.idxmax()
```

Out[24]:

```
41
```

In [25]:

```
# Same for type 2
# Except index different
Ba_series_type2 = glass_df['Ba'][47:98]
Ba_series_type2.idxmax()
```

Out[25]:

```
71
```

In [29]:

```
# Index 41 is max for type 1
glass_df = glass_df.drop(41, axis=0)
```

In [30]:

```
# index 71 is max for type 2
glass_df = glass_df.drop(71, axis=0)
```

In [31]:

```
# Drop index 109, max Ba for type 5
glass_df = glass_df.drop(109, axis=0)
```

Genearl comment: Not all chemical components are present in all glass types.
Type 6 does not contain any K.

Model prediction

Our first implemnentation. Assuming nonlinear relations.
Using a classifier with kernel to create a mapping to lower dimensionality before model training.

In [33]:

```
# Number of features to be plotted

num_vars = 9

# Create frame with features and types
# assign types to variable y
y_train = glass_df.iloc[:, num_vars]
# assign feature to variable X
X_train = glass_df.iloc[:, 0:num_vars]

# Test data
X_test = dfTest.iloc[:, 1:10]
```

In [34]:

```
# Make a pipeline using a kernel, SVC. Source book ch06

pipe_svc = make_pipeline(StandardScaler(),
                         KernelPCA(),
                         SVC(random_state=1))

# Tuning paramters contained in KernelPCA
# Need parameters for n_components and gamma.
# We try out two options for kernel: linear and rbf
# Gamma is by scikit learn ignored by linear kernel

components_range = [2, 4, 6, 8, 9]
gamma_range = [5, 10, 15, 20, 25]

# Construct parameter grid
param_grid = [{'kernelpca__n_components': components_range,
               'kernelpca__kernel': ['linear', 'rbf'],
               'kernelpca__gamma': gamma_range}]

gs = GridSearchCV(estimator=pipe_svc,
                  param_grid=param_grid,
                  scoring='accuracy',
                  cv=5,
                  refit=True,
                  n_jobs=-1)
# Perform grid search on training data
gs = gs.fit(X_train, y_train)
print(gs.best_score_)
print(gs.best_params_)
```

```
0.6357142857142857
{'kernelpca__gamma': 5, 'kernelpca__kernel': 'linear', 'kernelpca__n_compone
nts': 6}
```

Since the best kernel is linear the gamma parameter is not used.

Test if using kernel PCA improves are score.

In [35]:

```
from sklearn.svm import SVC
from sklearn.model_selection import cross_val_score


# Pipeline
pipe_svc = make_pipeline(StandardScaler(),
                         SVC(random_state=1))


# cross validation
scores = cross_val_score(estimator=pipe_svc,
                         X=X_train,
                         y=y_train,
                         cv=5,
                         n_jobs=1)
print('CV mean accuracy score: %s' % np.mean(scores))
```

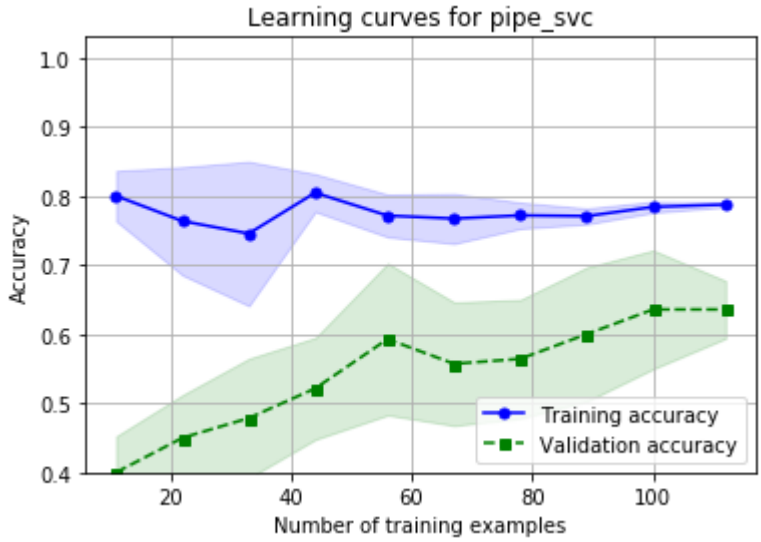
```
CV mean accuracy score: 0.6357142857142857
```

Using default values we achive the same result as with grid search and kernel PCA.

Now we debug our estimator using learning and validation curves.
This is useful to test how the model responds to increasing number of
training examples. And also to see if our model overfits

In [36]:

```
# AH note: need to shuffle the data
# Source: book ch06

train_sizes, train_scores, test_scores =\
                learning_curve(estimator=pipe_svc,
                               X=X_train,
                               y=y_train,
                               train_sizes=np.linspace(
                                           0.1, 1.0, 10),
                               cv=5,
                               n_jobs=-1,
                               shuffle=True,
                               random_state=1)
train_mean = np.mean(train_scores, axis=1)
train_std = np.std(train_scores, axis=1)
test_mean = np.mean(test_scores, axis=1)
test_std = np.std(test_scores, axis=1)

plt.plot(train_sizes, train_mean,
         color='blue', marker='o',
         markersize=5, label='Training accuracy')

plt.fill_between(train_sizes,
                 train_mean + train_std,
                 train_mean - train_std,
                 alpha=0.15, color='blue')

plt.plot(train_sizes, test_mean,
         color='green', linestyle='--',
         marker='s', markersize=5,
         label='Validation accuracy')
plt.fill_between(train_sizes,
                 test_mean + test_std,
                 test_mean - test_std,
                 alpha=0.15, color='green')

plt.grid()
plt.xlabel('Number of training examples')
plt.ylabel('Accuracy')
plt.legend(loc='lower right')
plt.ylim([0.4, 1.03])
plt.title('Learning curves for pipe_svc')
plt.show()
```



Comment: The gap between the curves is the closest at around 100. However the gap seems to be large, with a difference of around 0.15 points of accuracy at best. This is a sign that our model might be overfitting the data.

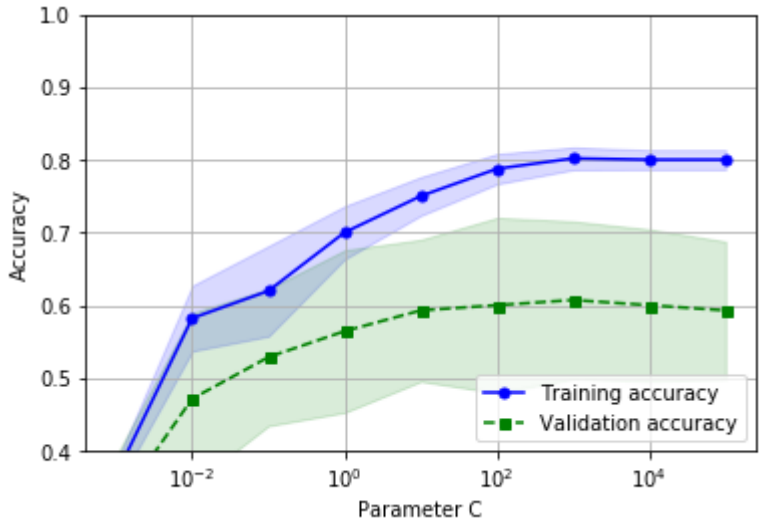
In [37]:

```
# Make a pipeline using regularisation

pipe_lr = make_pipeline(StandardScaler(),
                        LogisticRegression(random_state=1,
                                           max_iter=100000))

# List of regularisation parameters
param_range = [0.001, 0.01, 0.1, 1, 10, 100, 1000]
param_grid = [{'logisticregression__C': param_range}]

# Perform grid search
gs = GridSearchCV(estimator=pipe_lr,
                  param_grid=param_grid,
                  scoring='accuracy',
                  cv=5,
                  refit=True,
                  n_jobs=-1)
gs = gs.fit(X_train, y_train)
print(gs.best_score_)
print(gs.best_params_)
```

```
0.6071428571428571
{'logisticregression__C': 1000}
```

In [43]:

```
# Plot validation curve with using our best paramters
# in logistic regression estimator

pipe_lr = make_pipeline(StandardScaler(),
                        LogisticRegression(C=1000, random_state=1,
                                           max_iter=100000))

param_range = [0.001, 0.01, 0.1, 1, 10, 100, 1000,
               10000, 100000]
train_scores, test_scores = validation_curve(
                             estimator=pipe_lr,
                             X=X_train,
                             y=y_train,
                             param_name='logisticregression__C',
                             param_range=param_range,
                             cv=5)
train_mean = np.mean(train_scores, axis=1)
train_std = np.std(train_scores, axis=1)
test_mean = np.mean(test_scores, axis=1)
test_std = np.std(test_scores, axis=1)
plt.plot(param_range, train_mean,
         color='blue', marker='o',
         markersize=5, label='Training accuracy')
plt.fill_between(param_range, train_mean + train_std,
                 train_mean - train_std, alpha=0.15,
                 color='blue')
plt.plot(param_range, test_mean,
         color='green', linestyle='--',
         marker='s', markersize=5,
         label='Validation accuracy')
plt.fill_between(param_range,
                 test_mean + test_std,
                 test_mean - test_std,
                 alpha=0.15, color='green')
plt.grid()
plt.xscale('log')
plt.legend(loc='lower right')
plt.xlabel('Parameter C')
plt.ylabel('Accuracy')
plt.ylim([0.4, 1.0])
plt.show()
```



Comment: The best estimate for C is 1000, However our model seems to overfit the data with large gaps between training and validation accuracy. Similiar pattern as in learning curves plot for svc.

In [44]:

```
pipe_forest = make_pipeline(RandomForestClassifier(
                                random_state=1))

# Param range for number of trees and max depth
n_estimators_range = [10, 90, 100, 200]
depth_range = [1, 2, 3, 4, 5, 6, 7, None]

param_grid = [{'randomforestclassifier__n_estimators':
                n_estimators_range,
               'randomforestclassifier__max_depth':
               depth_range}]

gs_forest = GridSearchCV(estimator=pipe_forest,
                  param_grid=param_grid,
                  scoring='accuracy',
                  cv=5,
                  refit=True,
                  n_jobs=-1)

gs_forest = gs_forest.fit(X_train, y_train)
print(gs_forest.best_score_)
print(gs_forest.best_params_)
```

```
0.7071428571428572
{'randomforestclassifier__max_depth': 7, 'randomforestclassifier__n_estimato
rs': 90}
```

In [ ]:

```
# Predict using the best parameter combination
ypred = gs_forest.predict(X_test)
solution = pd.DataFrame(columns = ['label'], data = ypred)
solution['Id'] = solution.index
solution.to_csv('my_solution.csv', index=False)
```

Random forest has the highest accuracy. However its quite computationally heavy.

In [45]:

```
# Success criteria:

target_names = ['c1','c2','c3','c5','c6','c7']
print(classification_report(y_train,
                            cross_val_predict(
                                gs_forest,X_train,y_train,cv=5),
                            target_names=target_names))
```

```
C:\Users\ander\miniconda3\lib\site-packages\sklearn\model_selection\_split.p
y:670: UserWarning: The least populated class in y has only 4 members, which
is less than n_splits=5.
  warnings.warn(("The least populated class in y has only %d"
```

	precision	recall	f1-score	support
c1	0.63	0.63	0.63	46
c2	0.55	0.62	0.58	50
c3	0.57	0.36	0.44	11
c5	0.50	0.38	0.43	8
c6	0.43	0.50	0.46	6
c7	0.94	0.89	0.92	19
accuracy			0.62	140
macro avg	0.60	0.56	0.58	140
weighted avg	0.62	0.62	0.62	140

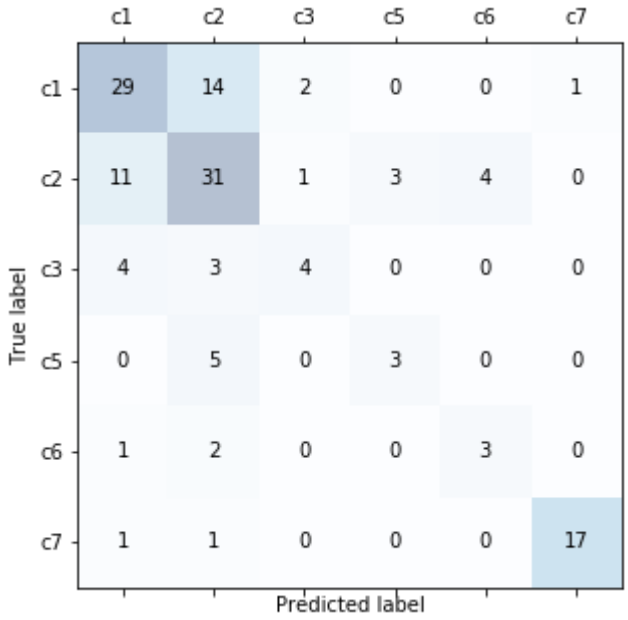
In [46]:

```
# Confusion matrix
# Source book: Python Machine Learning - ch06
# and solution DAT200 @kristl

# Construct confusion matrix
confmat = confusion_matrix(y_train, cross_val_predict(
                                gs_forest,X_train,y_train,cv=5))
```

```
C:\Users\ander\miniconda3\lib\site-packages\sklearn\model_selection\_split.p
y:670: UserWarning: The least populated class in y has only 4 members, which
is less than n_splits=5.
  warnings.warn(("The least populated class in y has only %d"
```

In [47]:

```
# Make plot of the confusion matrix

fig, ax = plt.subplots(figsize=(5, 5))
ax.matshow(confmat, cmap=plt.cm.Blues, alpha=0.3)
for i in range(confmat.shape[0]):
    for j in range(confmat.shape[1]):
        ax.text(x=j, y=i,
                s=confmat[i, j],
                va='center', ha='center')

plt.xticks(range(confmat.shape[1]), target_names)
plt.yticks(range(confmat.shape[0]), target_names)

plt.xlabel('Predicted label')
plt.ylabel('True label')
plt.show()
```



We can see that our estimator does a good job at predicting glass type 1, 2, 6 and 7. For type 3 and 5 half or less than half of the examples classified correctly.

Two class problem

We now distinguish only between float processed and non float processed classes

We will make a Receiver operating characteristic curve based on 5-fold cross-validation to see how well our model performs based on the False Positive Ratio (FPR) and the True Positive Ratio(TPR)

Combine class 1 and 3 and compare against class 2

In [48]:

```
1 group_type.count()
```

Out[48]:

```
type
1    47
2    51
3    11
5     9
6     6
7    19
Name: type, dtype: int64
```

In [49]:

```
# Last index 108 which the 107th example
# Make a new df with float processed and
# non float processed glass
df2class = glass_df.loc[:108 ,]

# Make a new column
df2class['float_processed'] = np.where(df2class['type']!=2,
                                       1, 0)
df2class
```

```
<ipython-input-49-5525686aeb25>:7: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/
stable/user_guide/indexing.html#returning-a-view-versus-a-copy (https://pand
as.pydata.org/pandas-docs/stable/user_guide/indexing.html#returning-a-view-v
ersus-a-copy)
  df2class['float_processed'] = np.where(df2class['type']!=2,
```

Out[49]:

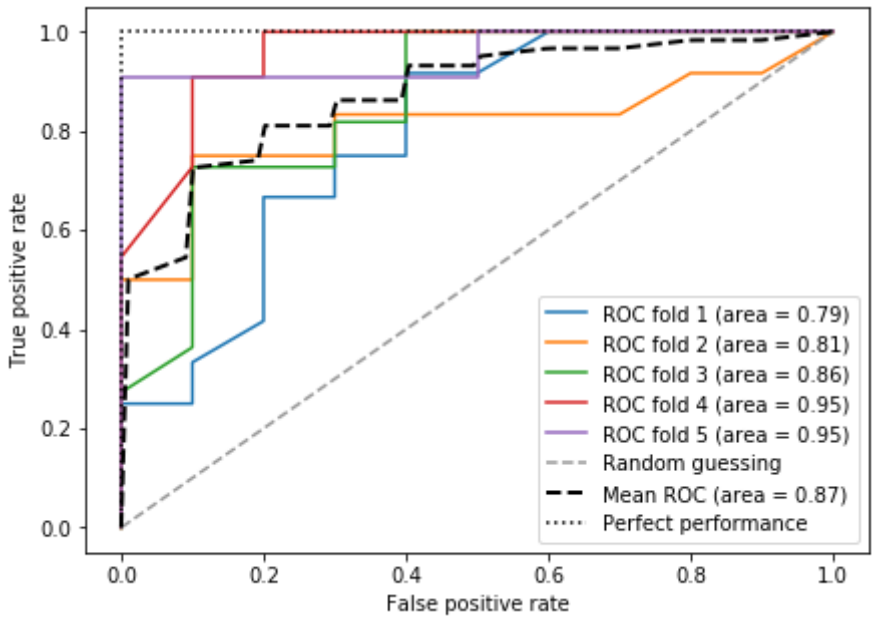
	RI	Na	Mg	Al	Si	K	Ca	Ba	Fe	type	float_processed
0	1.52101	13.64	4.49	1.10	71.78	0.06	8.75	0.0	0.00	1	1
1	1.51761	13.89	3.60	1.36	72.73	0.48	7.83	0.0	0.00	1	1
2	1.51766	13.21	3.69	1.29	72.61	0.57	8.22	0.0	0.00	1	1
3	1.51742	13.27	3.62	1.24	73.08	0.55	8.07	0.0	0.00	1	1
4	1.51743	13.30	3.60	1.14	73.09	0.58	8.17	0.0	0.00	1	1
...	...	...	...	...	...	...	...	...	...	...	...
104	1.51655	13.41	3.39	1.28	72.64	0.52	8.65	0.0	0.00	3	1
105	1.52121	14.03	3.76	0.58	71.79	0.11	9.65	0.0	0.00	3	1
106	1.51796	13.50	3.36	1.63	71.94	0.57	8.81	0.0	0.09	3	1
107	1.51832	13.33	3.34	1.54	72.14	0.56	8.99	0.0	0.00	3	1
108	1.52211	14.19	3.78	0.91	71.36	0.23	9.14	0.0	0.37	3	1

107 rows × 11 columns

In [50]:

```
# Source Pyhon Machine Learning - Ch06

# Using the best parameters from our best classifier random_forest

# Number of features to be plotted

num_vars = 9

# Create frame with features and types
# assign class to variable y
y_train2 = df2class.iloc[:, -1].values
# assign feature to variable X
X_train2 = df2class.iloc[:, 0:num_vars].values

# List of indices in training and test data
# for different splits
# Shuffle within each class before splitting to mix
# type 1 and type 3
cv = list(StratifiedKFold(n_splits=5,
                          shuffle=True,
                          random_state=1).split(X_train2,
                                                y_train2))

fig = plt.figure(figsize=(7, 5))

mean_tpr = 0.0
mean_fpr = np.linspace(0, 1, 100)
all_tpr = []

for i, (train, test) in enumerate(cv):
    # First fit data with training set then predict
    # probabilities of the two classes on the test set
    probas = pipe_forest.fit(
        X_train2[train],
        y_train2[train]).predict_proba(X_train2[test])
    fpr, tpr, thresholds = roc_curve(y_train2[test],
                                     probas[:, 1],
                                     pos_label=1)
    mean_tpr += np.interp(mean_fpr, fpr, tpr)
    mean_tpr[0] = 0.0
    roc_auc = auc(fpr, tpr)
    plt.plot(fpr,
             tpr,
             label='ROC fold %d (area = %0.2f)'
             % (i+1, roc_auc))
plt.plot([0, 1],
         [0, 1],
         linestyle='--',
         color=(0.6, 0.6, 0.6),
         label='Random guessing')

mean_tpr /= len(cv)
mean_tpr[-1] = 1.0
mean_auc = auc(mean_fpr, mean_tpr)
plt.plot(mean_fpr, mean_tpr, 'k--',
         label='Mean ROC (area = %0.2f)' % mean_auc, lw=2)
plt.plot([0, 0, 1],
         [0, 1, 1],
         linestyle=':',
```

```
         color='black',
         label='Perfect performance')
plt.xlim([-0.05, 1.05])
plt.ylim([-0.05, 1.05])
plt.xlabel('False positive rate')
plt.ylabel('True positive rate')
plt.legend(loc='lower right')
plt.show()

```



The performance is here measured as the ROC area under the curve (ROC AUC). Perfect performance is represented by the curve going vertical and then horisontal after reaching a TPR of 1. This has a ROC AUC of 1. While the 45-degree dashed line corresponds to Rdandom guessing and has a ROC AUC of 0.5.

Our classifier has a mean value of 0.87. Well above the threshold for random guessing. However interestlingly the ROC curves at different folds varies a lot. Might be because of the relatively few examples in our training set.

In [ ]:

```
1
```