Glass type classification with machine learning

This is my first Kaggle notebook. Here's my plan of attack for the glass classification problem.

Contents

1) Prepare Problem

- Load libraries
- Load and explore the shape of the dataset

2) Summarize Data

- Descriptive statistics
- Data visualization

3) Prepare Data

- Data Cleaning
- Split-out validation dataset
- Data transformation

4) Evaluate Algorithms

- Assessing feature importance via XGBoost and PCA
- Compare Algorithms

5) Improve Accuracy

- Algorithm Tuning

6) Finalize Model

- Create standalone model on entire training dataset
- Predictions on test dataset

1. Prepare Problem

Loading the libraries

Let us first begin by loading the libraries that we'll use in the notebook

In [1]:
```
import numpy as np  # linear algebra
import pandas as pd  # read dataframes
import matplotlib.pyplot as plt # visualization
import seaborn as sns # statistical visualizations and aesthetics
from sklearn.preprocessing import StandardScaler # preprocessing
from sklearn.decomposition import PCA # dimensionality reduction
from scipy.stats import boxcox # data transform
from sklearn.model_selection import (train_test_split, KFold , cros
s_val_score, GridSearchCV ) # model selection modules
from sklearn.pipeline import Pipeline # streaming pipelines
# load models
from sklearn.tree import DecisionTreeClassifier
from xgboost import (XGBClassifier, plot_importance)
from sklearn.svm import SVC
from sklearn.ensemble import (RandomForestClassifier, AdaBoostClass
ifier)
from sklearn.neighbors import KNeighborsClassifier
from sklearn.naive_bayes import GaussianNB
%matplotlib inline
```

Loading and exploring the shape of the dataset

In [2]:
```
df = pd.read_csv('../input/glass.csv')

print(df.shape)
```

```
(214, 10)
```

The dataset consists of 214 observations

In [3]:
```
df.head(15)
```

Out[3]:

	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	0.0	0.00	1
1	1.51761	13.89	3.60	1.36	72.73	0.48	7.83	0.0	0.00	1
2	1.51618	13.53	3.55	1.54	72.99	0.39	7.78	0.0	0.00	1
3	1.51766	13.21	3.69	1.29	72.61	0.57	8.22	0.0	0.00	1
4	1.51742	13.27	3.62	1.24	73.08	0.55	8.07	0.0	0.00	1
5	1.51596	12.79	3.61	1.62	72.97	0.64	8.07	0.0	0.26	1
6	1.51743	13.30	3.60	1.14	73.09	0.58	8.17	0.0	0.00	1
7	1.51756	13.15	3.61	1.05	73.24	0.57	8.24	0.0	0.00	1
8	1.51918	14.04	3.58	1.37	72.08	0.56	8.30	0.0	0.00	1
9	1.51755	13.00	3.60	1.36	72.99	0.57	8.40	0.0	0.11	1
10	1.51571	12.72	3.46	1.56	73.20	0.67	8.09	0.0	0.24	1
11	1.51763	12.80	3.66	1.27	73.01	0.60	8.56	0.0	0.00	1
12	1.51589	12.88	3.43	1.40	73.28	0.69	8.05	0.0	0.24	1
13	1.51748	12.86	3.56	1.27	73.21	0.54	8.38	0.0	0.17	1
14	1.51763	12.61	3.59	1.31	73.29	0.58	8.50	0.0	0.00	1

In [4]:
```
df.dtypes
```

Out[4]:
```
RI      float64
Na      float64
Mg      float64
Al      float64
Si      float64
K       float64
Ca      float64
Ba      float64
Fe      float64
Type      int64
dtype: object
```

2. Summarize data

Descriptive statistics

Let's first summarize the distribution of the numerical variables.

In [5]:
```
df.describe()
```

Out[5]:

	RI	Na	Mg	Al	Si	K	Ca
count	214.000000	214.000000	214.000000	214.000000	214.000000	214.000000	21
mean	1.518365	13.407850	2.684533	1.444907	72.650935	0.497056	8.
std	0.003037	0.816604	1.442408	0.499270	0.774546	0.652192	1.
min	1.511150	10.730000	0.000000	0.290000	69.810000	0.000000	5.
25%	1.516523	12.907500	2.115000	1.190000	72.280000	0.122500	8.
50%	1.517680	13.300000	3.480000	1.360000	72.790000	0.555000	8.
75%	1.519157	13.825000	3.600000	1.630000	73.087500	0.610000	9.
max	1.533930	17.380000	4.490000	3.500000	75.410000	6.210000	16

The features are not on the same scale. For example Si has a mean of 72.65 while Fe has a mean value of 0.057. Features should be on the same scale for an algorithm such as logistic regression (gradient descent) to converge fast. Let's go ahead and check the distribution of the glass types.

In [6]:
```
df['Type'].value_counts()
```

Out[6]:
```
2    76
1    70
7    29
3    17
5    13
6     9
Name: Type, dtype: int64
```

The dataset is pretty unbalanced. The instances of types 1 and 2 constitute more than 67 % of the glass types.

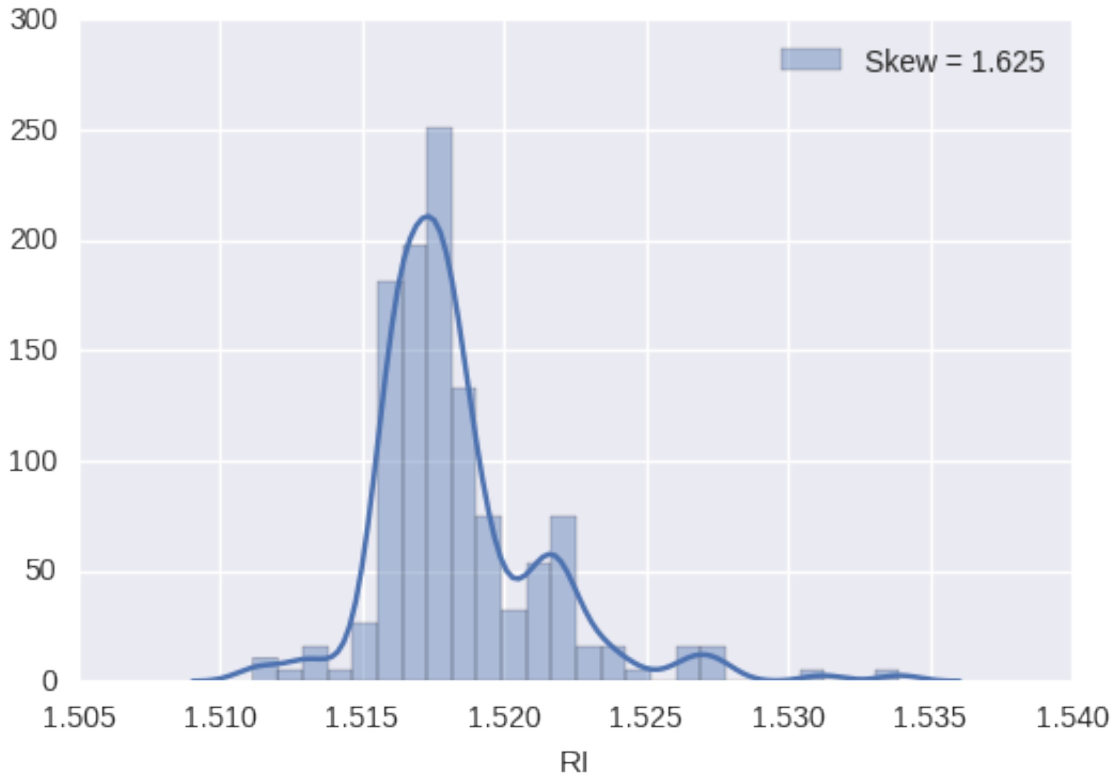
Data Visualization

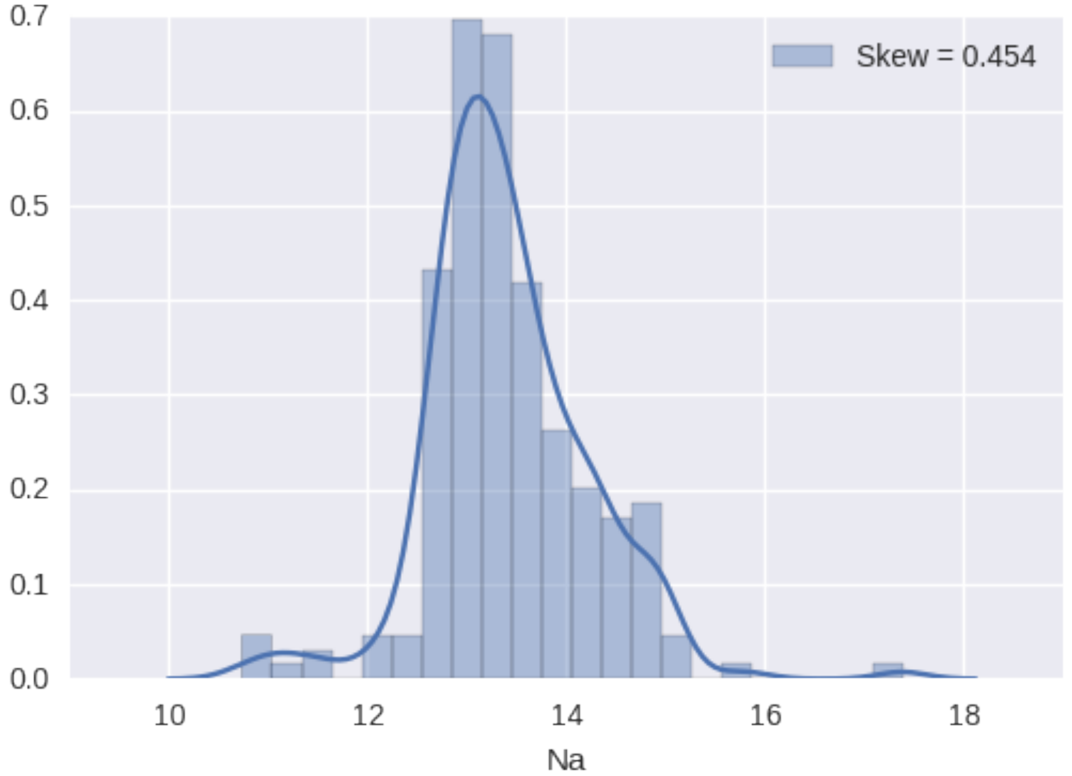
- **Univariate plots**

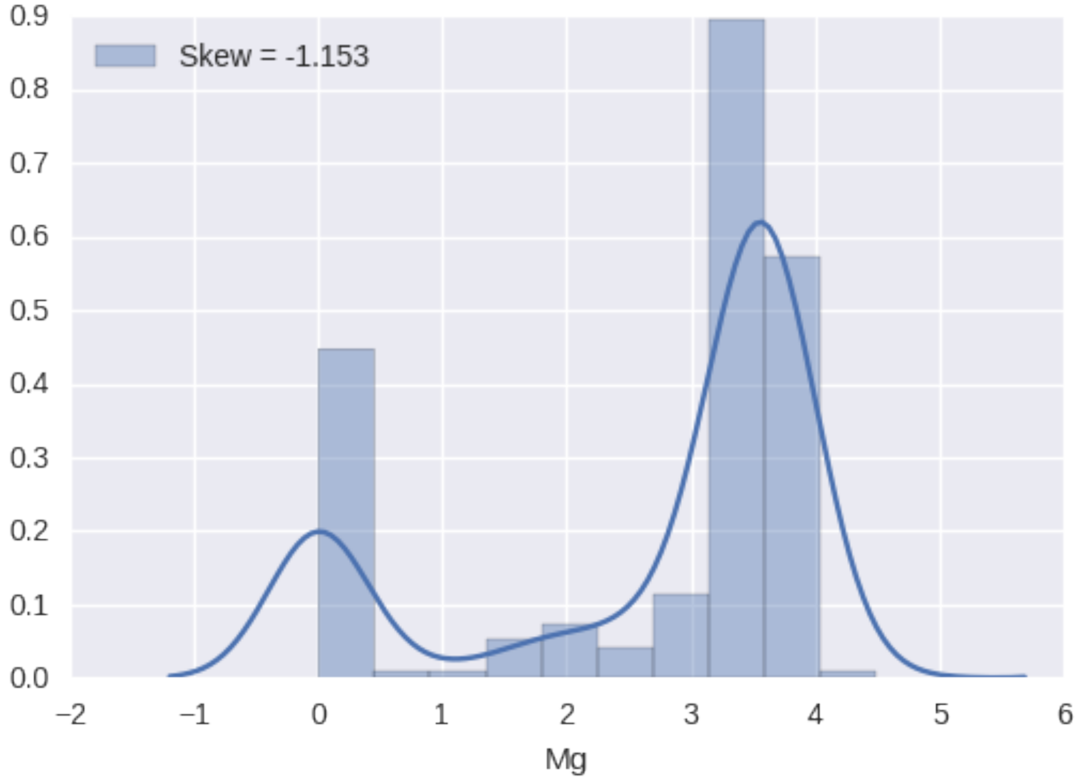
Let's go ahead an look at the distribution of the different features of this dataset.

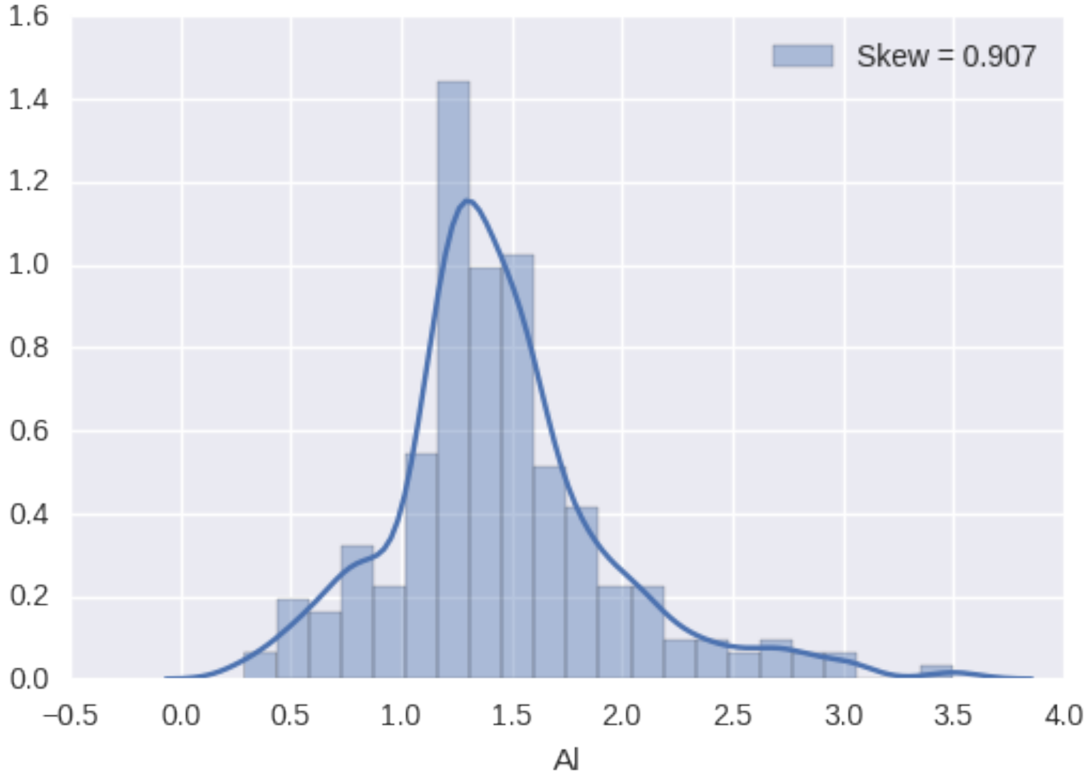
In [7]:
```
features = df.columns[:-1].tolist()
for feat in features:
    skew = df[feat].skew()
    sns.distplot(df[feat], label='Skew = %.3f' %(skew))
    plt.legend(loc='best')
    plt.show()
```

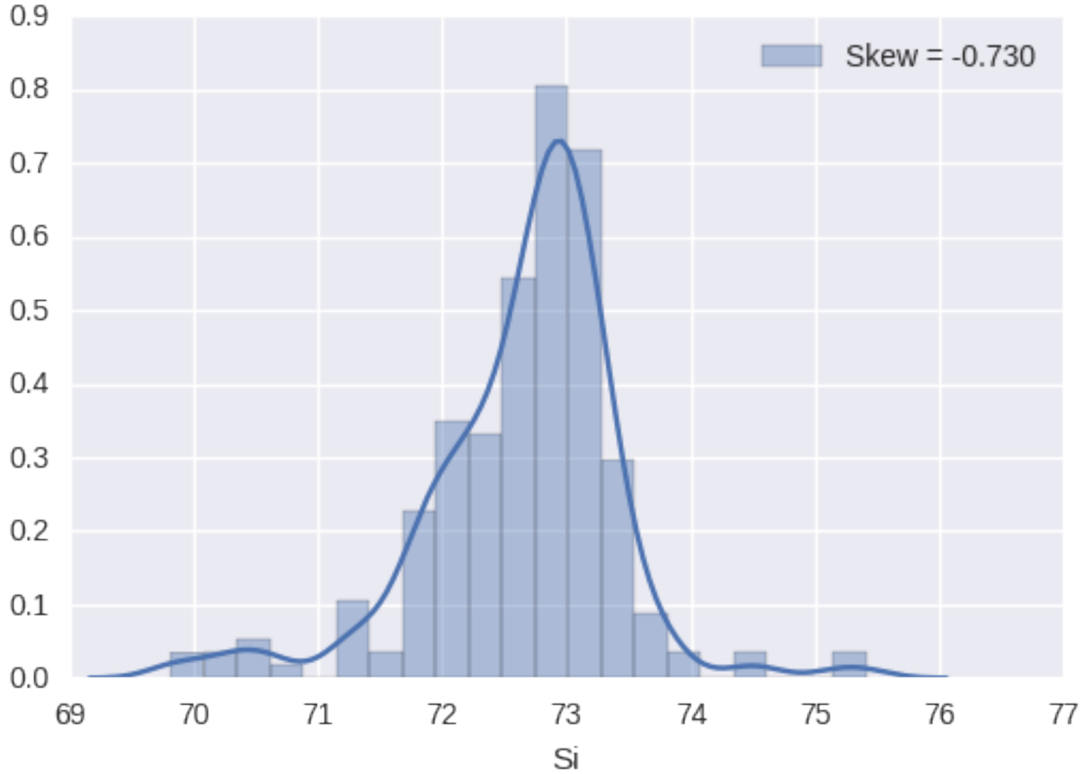
```
/opt/conda/lib/python3.5/site-packages/statsmodels/nonparametric/k
detools.py:20: VisibleDeprecationWarning: using a non-integer numb
er instead of an integer will result in an error in the future
  y = X[:m/2+1] + np.r_[0,X[m/2+1:],0]*1j
```

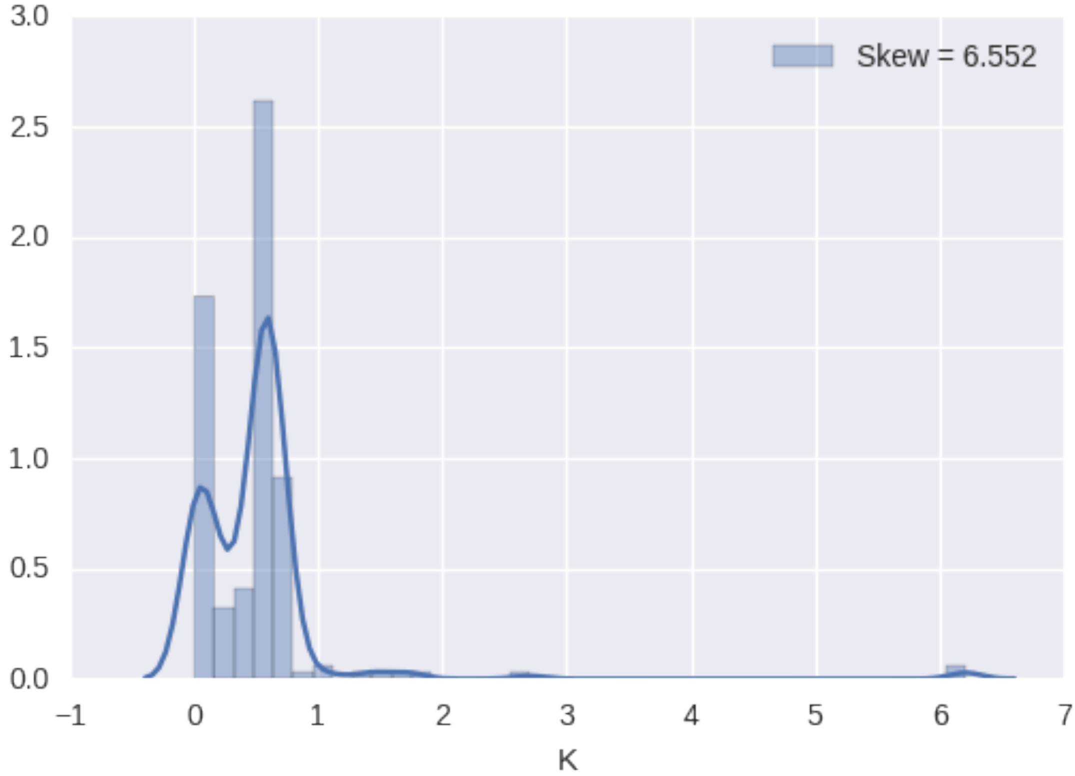


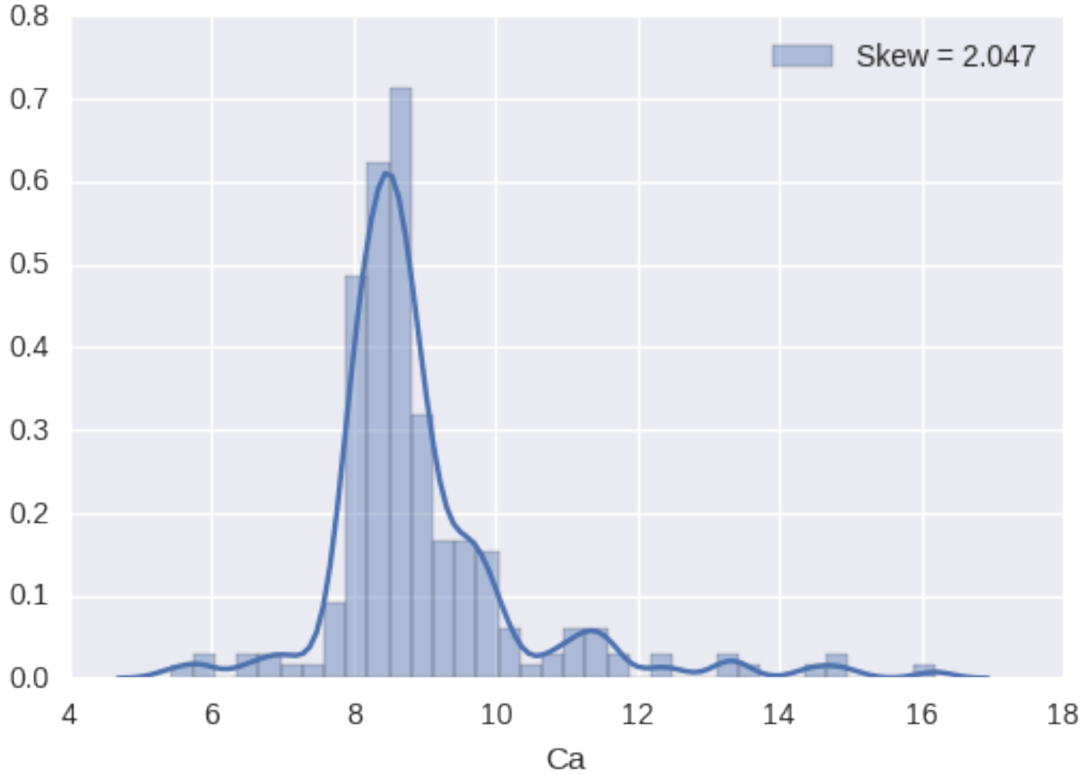

Skew = 0.454
0.0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
10
12
14
16
18
Na

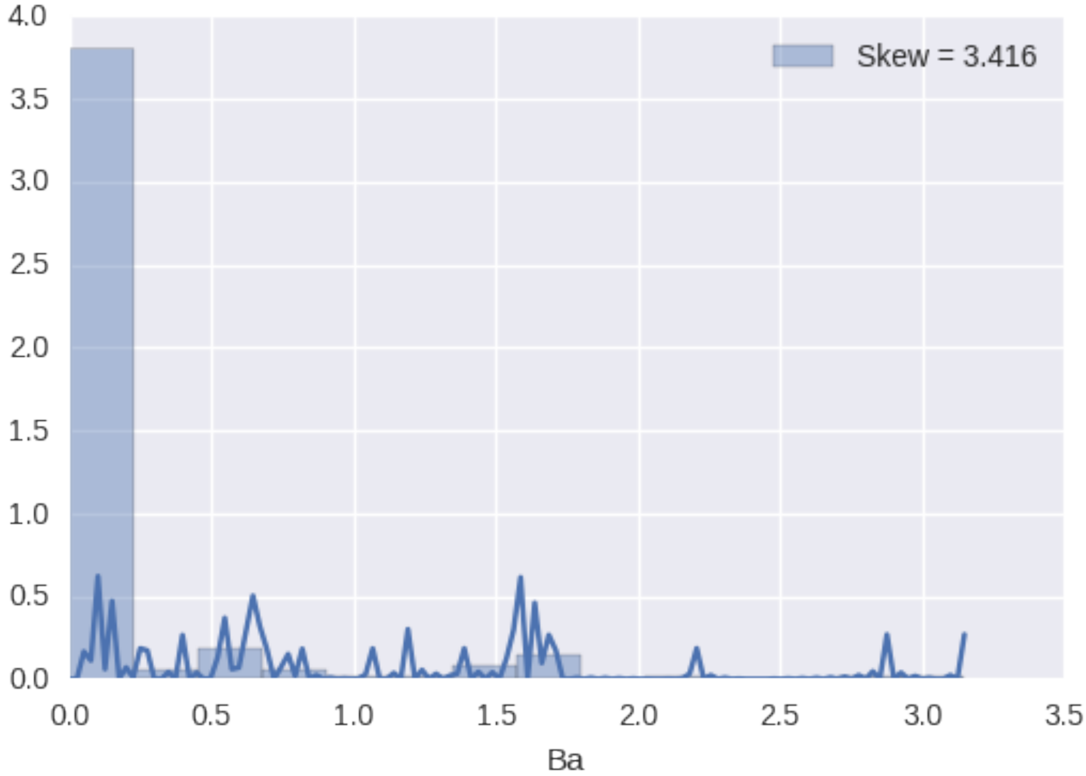

Skew = -1.153
0.0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
−2
−1
0
1
2
3
4
5
6
Mg

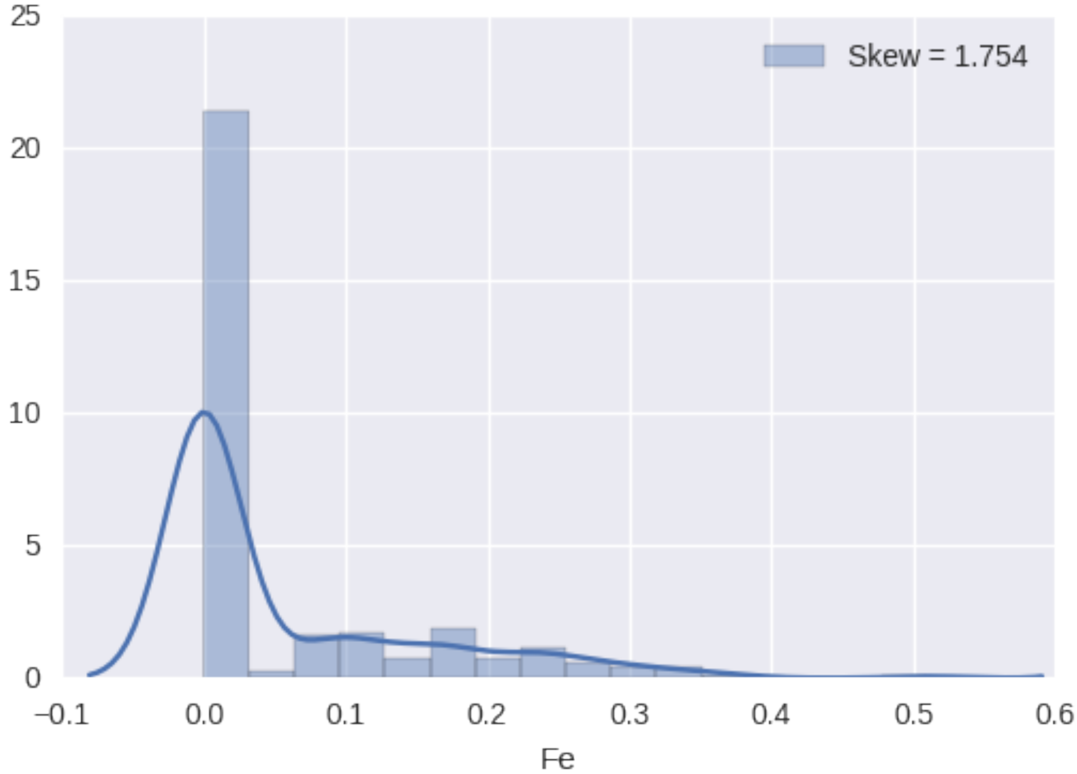

Skew = 0.907
1.6
1.4
1.2
1.0
0.8
0.6
0.4
0.2
0.0
−0.5
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
4.0
Al


Skew = -0.730
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0.0
69
70
71
72
73
74
75
76
77
Si


Skew = 6.552
K


Skew = 2.047
Ca

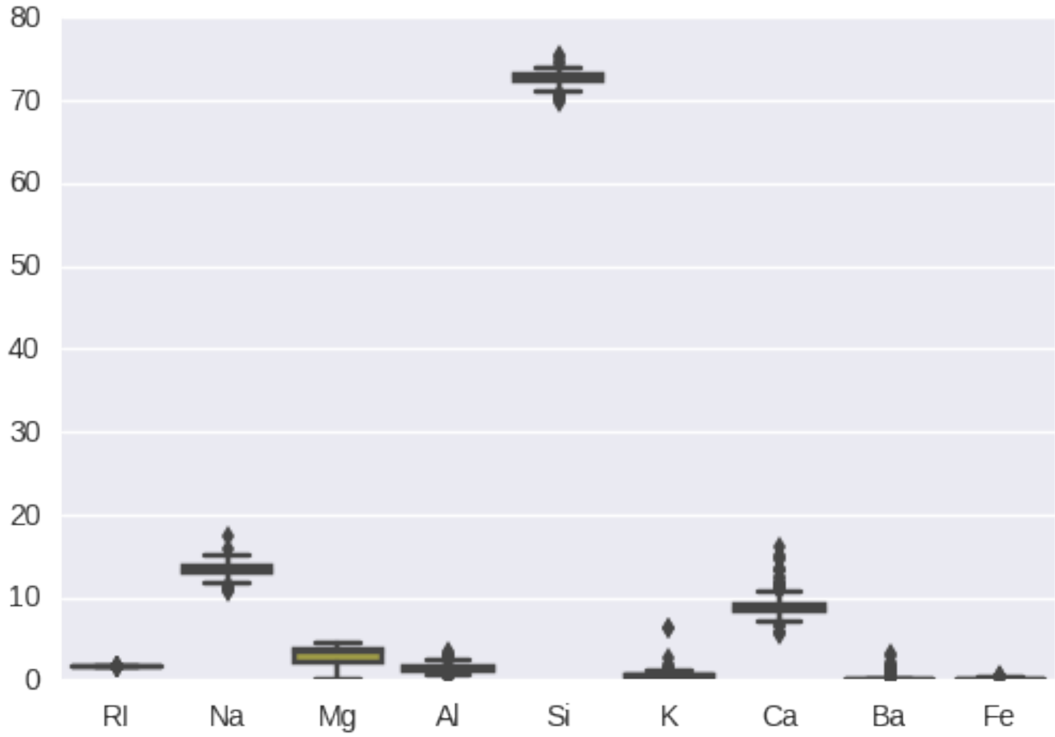




None of the features is normally distributed. The features Fe, Ba, Ca and K exhibit the highest skew coefficients. Let's do a boxplot of the several distributions.

In [8]:
```
sns.boxplot(df[features])
plt.show()
```

```
/opt/conda/lib/python3.5/site-packages/seaborn/categorical.py:2171
: UserWarning: The boxplot API has been changed. Attempting to adj
ust your arguments for the new API (which might not work). Please
update your code. See the version 0.6 release notes for more info.
  warnings.warn(msg, UserWarning)
```



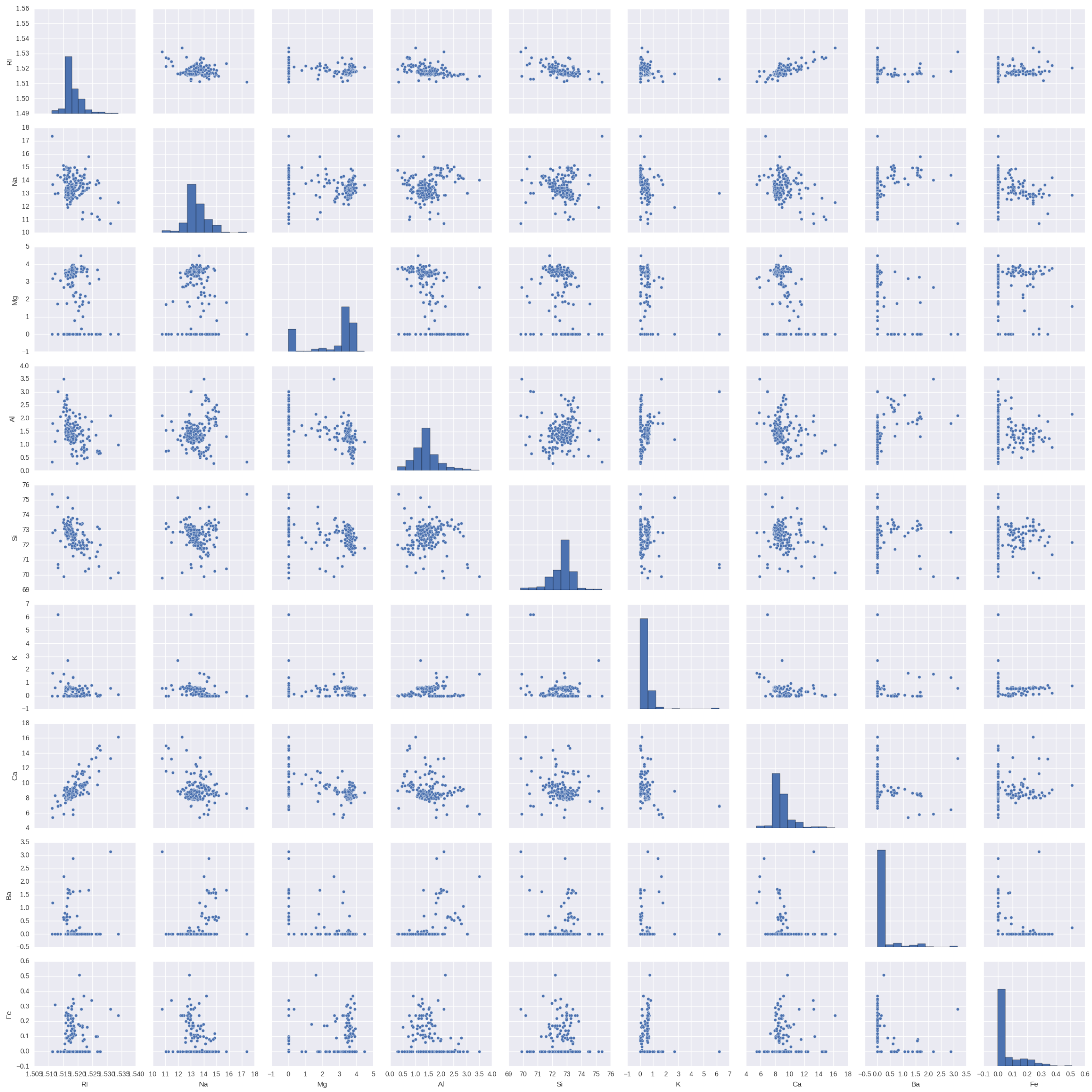
Unsurprisingly, Silicon has a mean that is much superior to the other constituents as we already saw in the previous section. Well, that is normal since glass is mainly based on silica.

- **Multivariate plots**

Let's now do a pairplot to visually examine the correlation between the features.

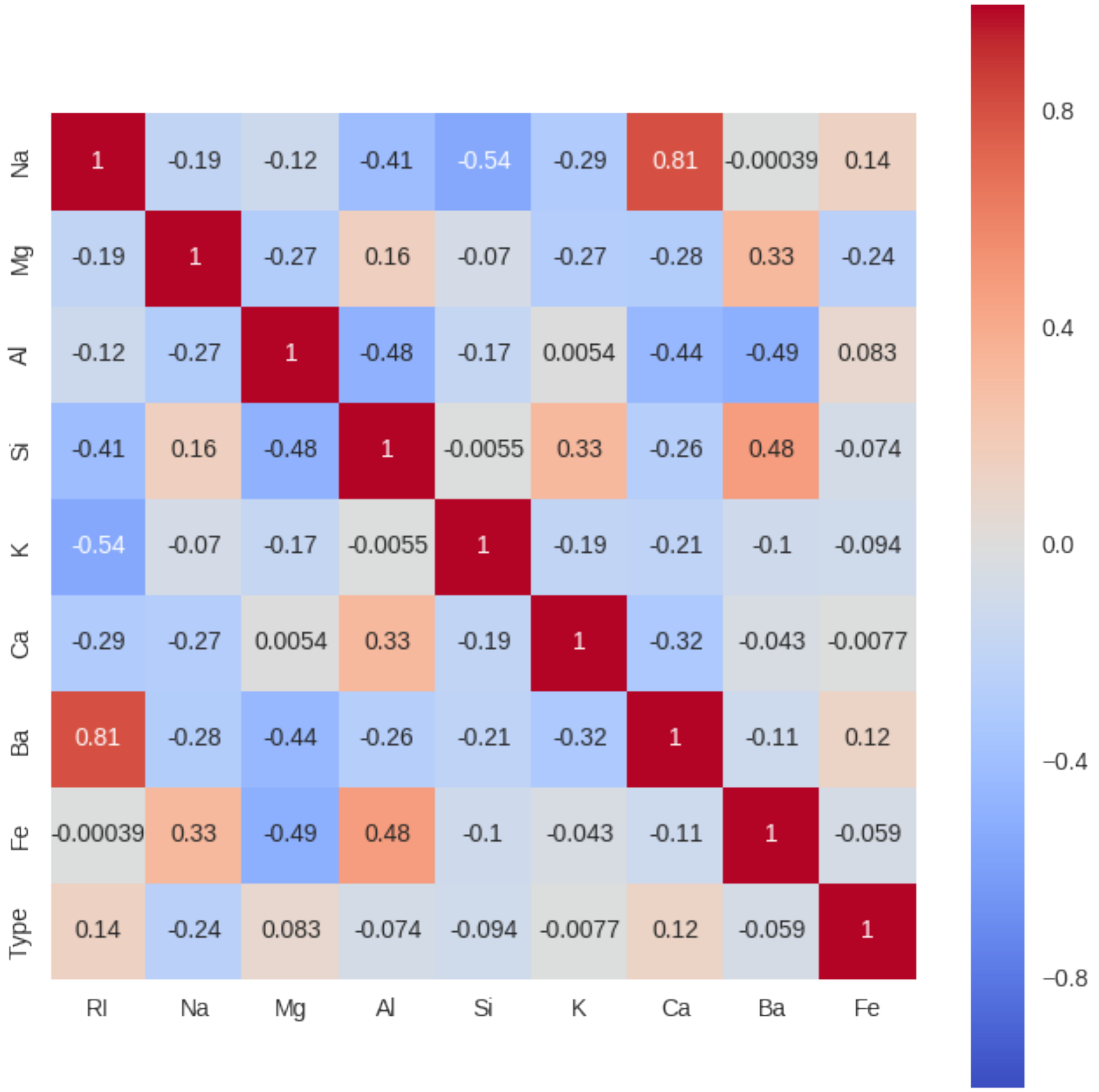
In [9]:
```
plt.figure(figsize=(8,8))
sns.pairplot(df[features],palette='coolwarm')
plt.show()
```

```
<matplotlib.figure.Figure at 0x7f1a21818518>
```



Let's go ahead and examine a heatmap of the correlations.

In [10]:
```
corr = df[features].corr()
plt.figure(figsize=(8,8))
sns.heatmap(corr, cbar = True,  square = True, annot=True,
           xticklabels= df.columns.tolist(), yticklabels= df.column
s.tolist(),
           cmap= 'coolwarm')
plt.show()
print(corr)
```


Na
Mg
Al
Si
K
Ca
Ba
Fe
Type
RI
Na
Mg
Al
Si
K
Ca
Ba
Fe
1
-0.19
-0.12
-0.41
-0.54
-0.29
0.81
-0.00039
0.14
-0.19
1
-0.27
0.16
-0.07
-0.27
-0.28
0.33
-0.24
-0.12
-0.27
1
-0.48
-0.17
0.0054
-0.44
-0.49
0.083
-0.41
0.16
-0.48
1
-0.0055
0.33
-0.26
0.48
-0.074
-0.54
-0.07
-0.17
-0.0055
1
-0.19
-0.21
-0.1
-0.094
-0.29
-0.27
0.0054
0.33
-0.19
1
-0.32
-0.043
-0.0077
0.81
-0.28
-0.44
-0.26
-0.21
-0.32
1
-0.11
0.12
-0.00039
0.33
-0.49
0.48
-0.1
-0.043
-0.11
1
-0.059
0.14
-0.24
0.083
-0.074
-0.094
-0.0077
0.12
-0.059
1
0.8
0.4
0.0
−0.4
−0.8

```
          RI        Na        Mg        Al        Si         K
Ca  \
RI  1.000000 -0.191885 -0.122274 -0.407326 -0.542052 -0.289833  0.
810403
Na -0.191885  1.000000 -0.273732  0.156794 -0.069809 -0.266087 -0.
275442
Mg -0.122274 -0.273732  1.000000 -0.481799 -0.165927  0.005396 -0.
443750
Al -0.407326  0.156794 -0.481799  1.000000 -0.005524  0.325958 -0.
259592
Si -0.542052 -0.069809 -0.165927 -0.005524  1.000000 -0.193331 -0.
208732
K  -0.289833 -0.266087  0.005396  0.325958 -0.193331  1.000000 -0.
317836
Ca  0.810403 -0.275442 -0.443750 -0.259592 -0.208732 -0.317836  1.
000000
Ba -0.000386  0.326603 -0.492262  0.479404 -0.102151 -0.042618 -0.
112841
Fe  0.143010 -0.241346  0.083060 -0.074402 -0.094201 -0.007719  0.
124968

          Ba        Fe
RI -0.000386  0.143010
Na  0.326603 -0.241346
Mg -0.492262  0.083060
Al  0.479404 -0.074402
Si -0.102151 -0.094201
K  -0.042618 -0.007719
Ca -0.112841  0.124968
Ba  1.000000 -0.058692
Fe -0.058692  1.000000
```

There seems to be a strong positive correlation between RI and Ba; also a strong positive correlation between Ba and Na is noticeable. This could give us a hint about performing Principal component analysis to decorrelate some of the input features.

3. Prepare data

- Data cleaning

In [11]:
```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 214 entries, 0 to 213
Data columns (total 10 columns):
RI      214 non-null float64
Na      214 non-null float64
Mg      214 non-null float64
Al      214 non-null float64
Si      214 non-null float64
K       214 non-null float64
Ca      214 non-null float64
Ba      214 non-null float64
Fe      214 non-null float64
Type    214 non-null int64
dtypes: float64(9), int64(1)
memory usage: 16.8 KB
```

This dataset is clean; there aren't any missing values in it.

- Split-out validation dataset

In [12]:
```
# Define X as features and y as lablels
X = df[features]
y = df['Type']
# set a seed and a test size for splitting the dataset
seed = 7
test_size = 0.2

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size
= test_size , random_state = seed)
```

- Data transformation

Let's examine if a Box-Cox transform can contribute to the normalization of some features. It should be emphasized that all transformations should only be done on the training set to avoid data snooping. Otherwise the test error estimation will be biased.

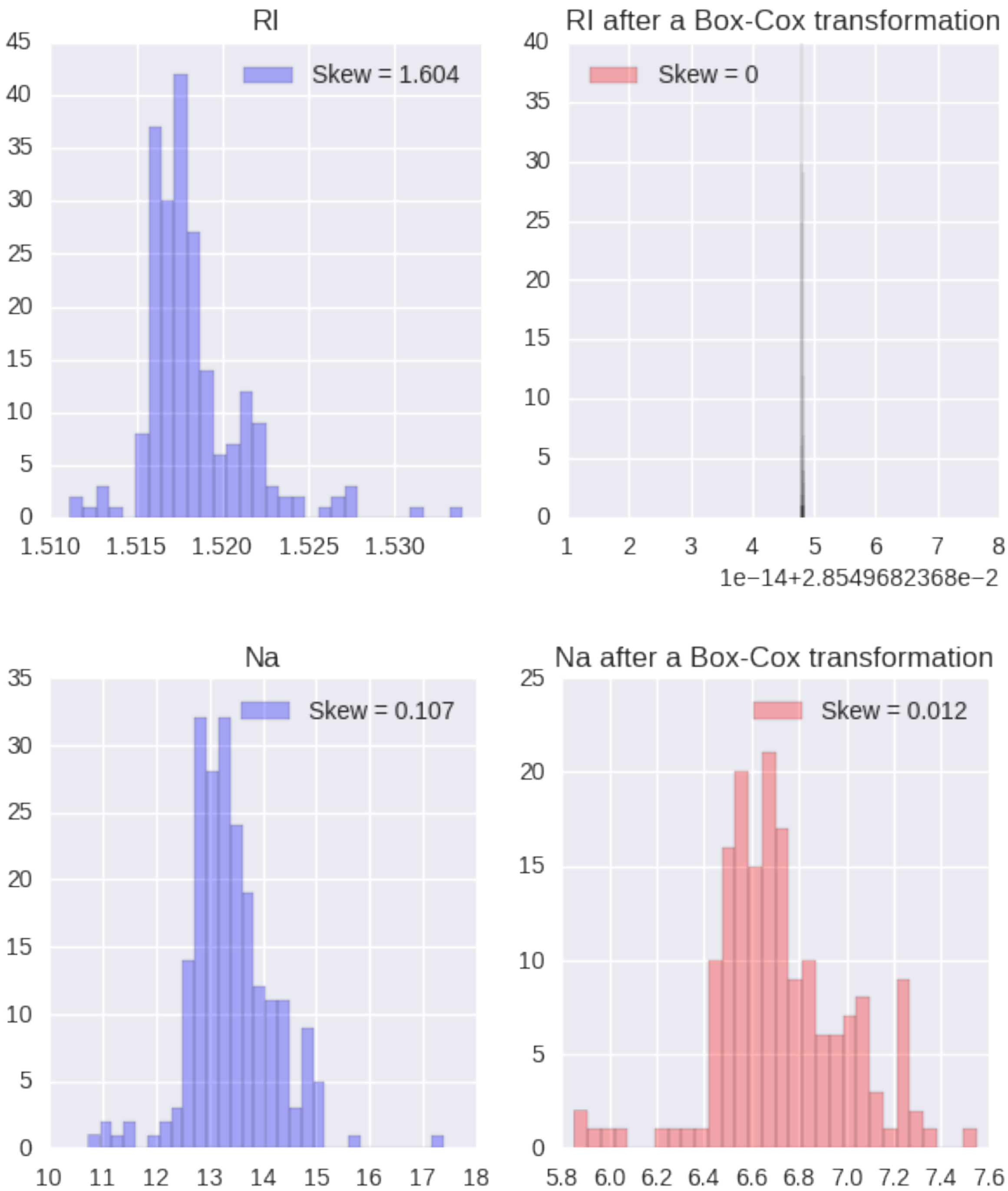
In [13]:
```
features_boxcox = []

for feature in features:
    bc_transformed, _ = boxcox(X_train[feature]+1)  # shift by 1 to
avoid computing log of negative values
    features_boxcox.append(bc_transformed)

features_boxcox = np.column_stack(features_boxcox)
df_bc = pd.DataFrame(data=features_boxcox, columns=features)
df_bc['Type'] = df['Type']
```

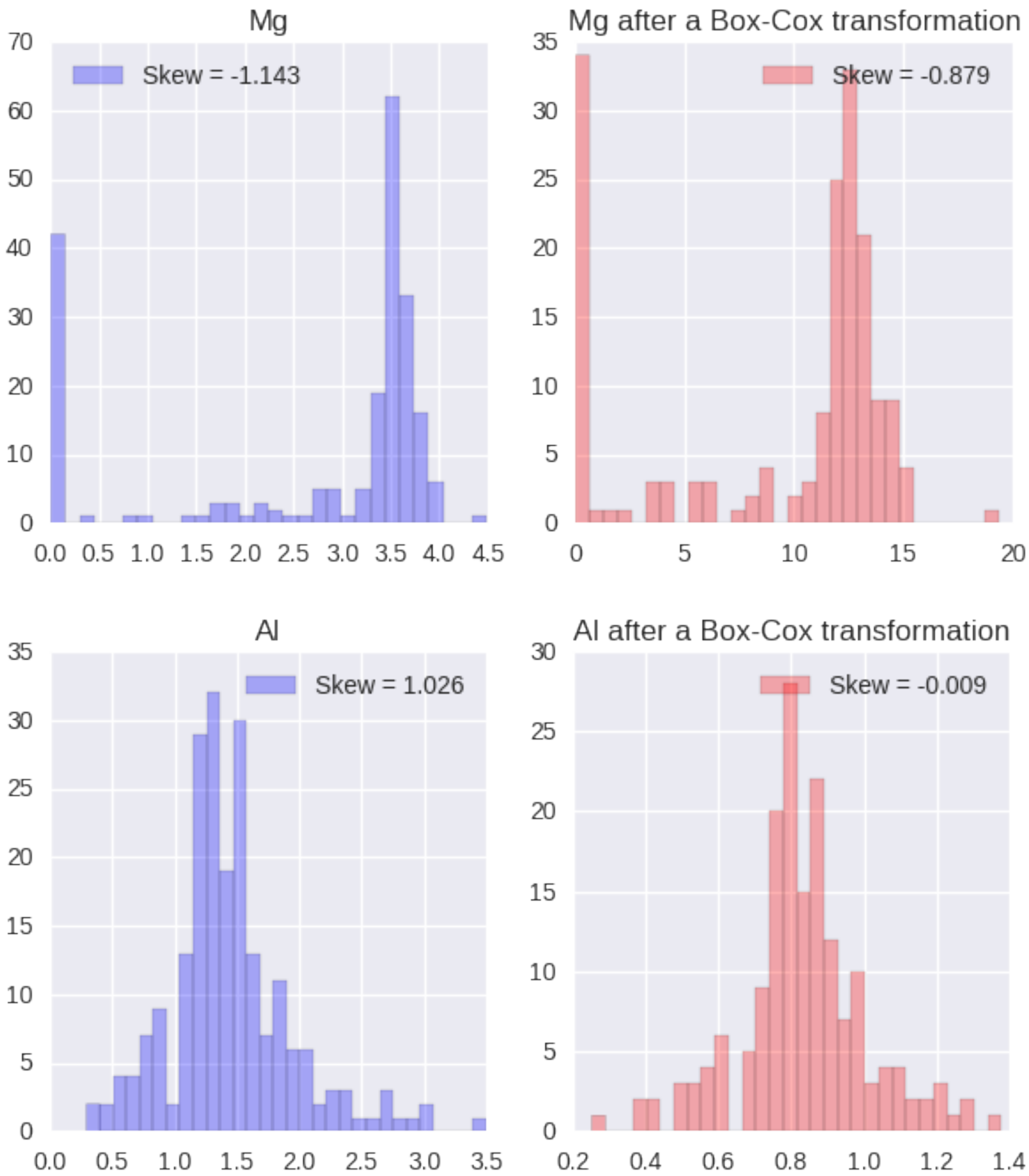
In [14]:
```
df_bc.head()
```

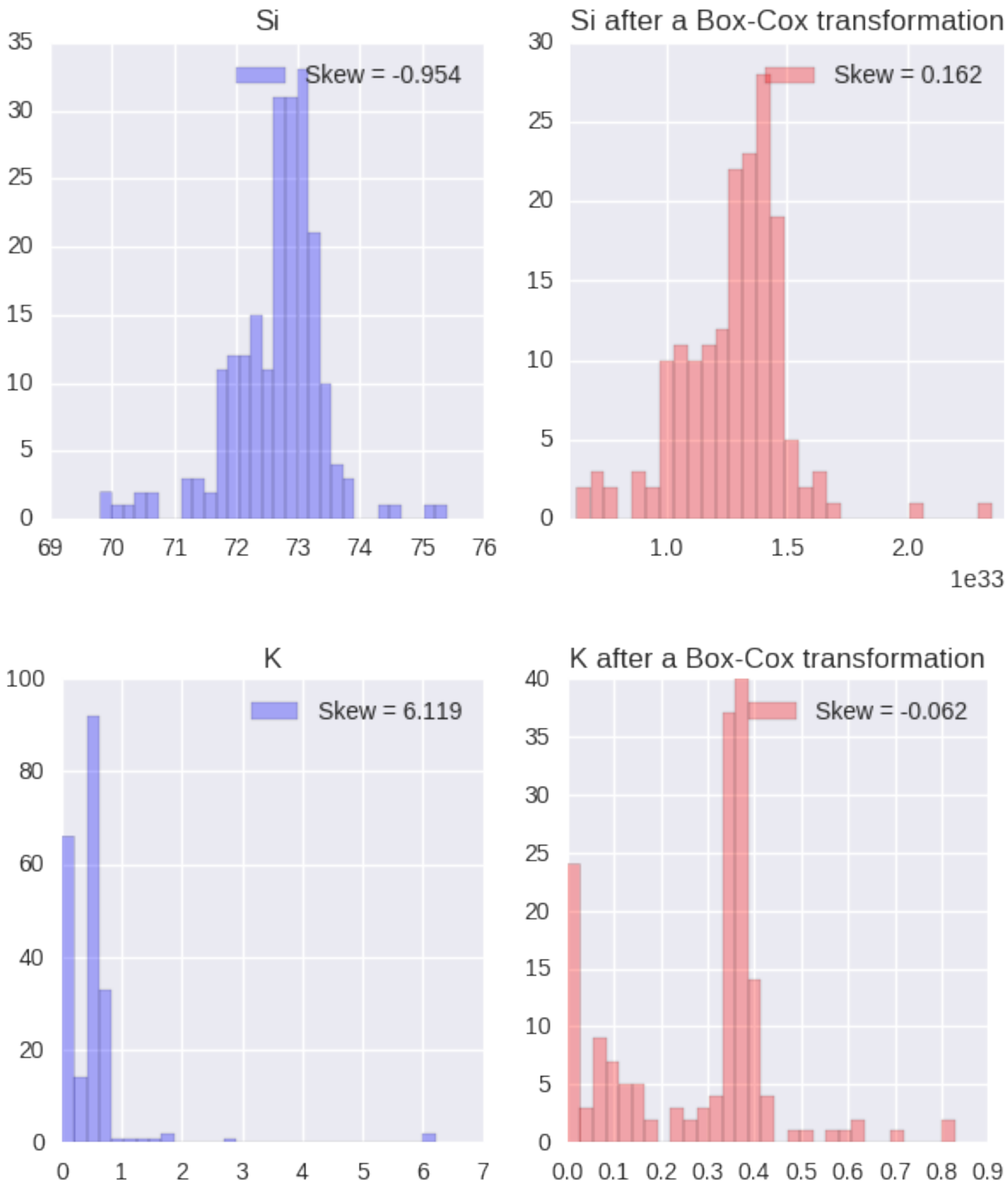
Out[14]:

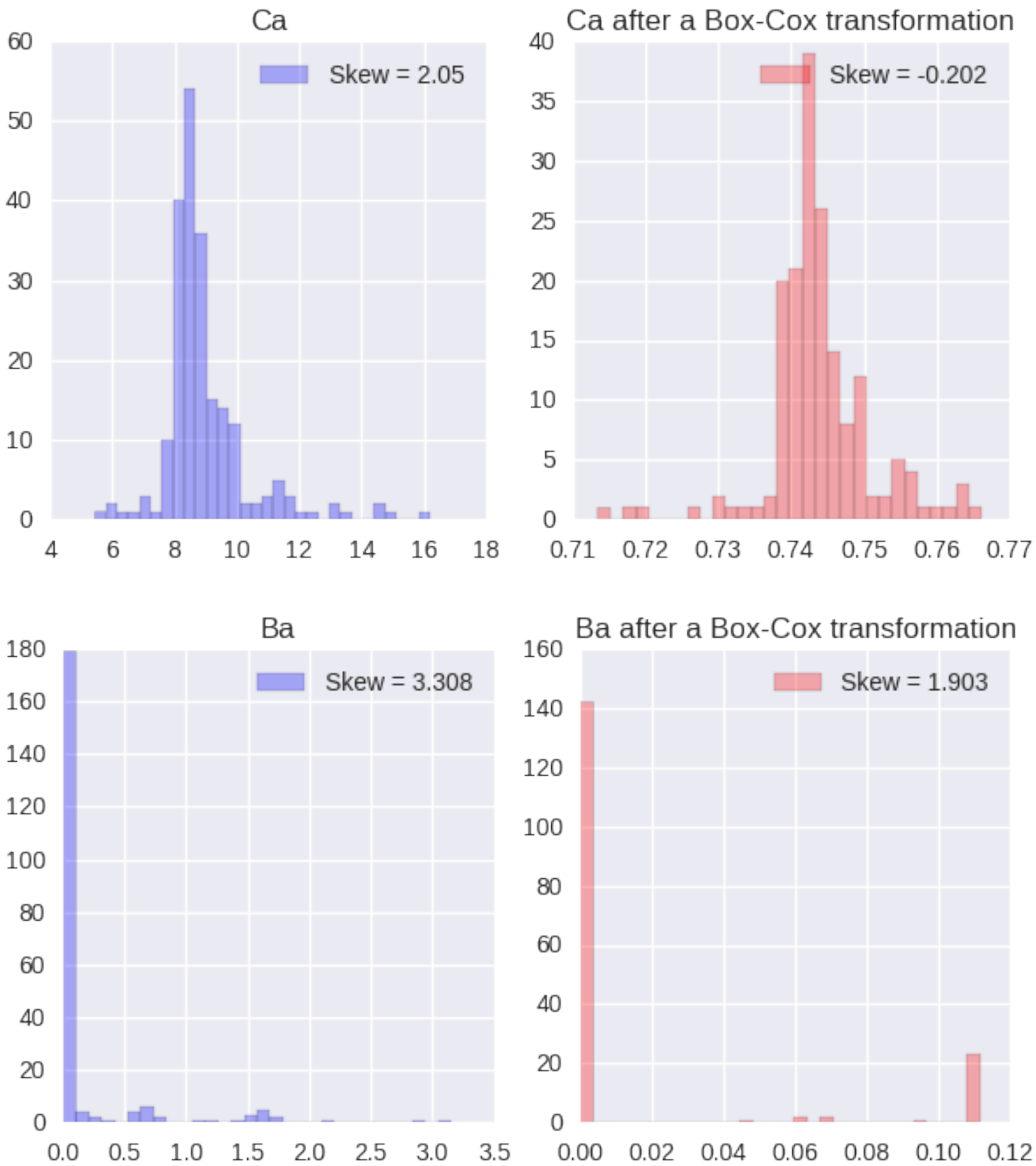
	RI	Na	Mg	Al	Si	K	Ca	Ba
0	0.02855	6.837366	15.163991	0.883137	1.023264e+33	0.346637	0.740489	0.000(
1	0.02855	6.559483	13.326088	0.890167	1.220312e+33	0.362661	0.742602	0.000(
2	0.02855	7.335804	0.000000	1.101006	1.561008e+33	0.000000	0.741317	0.110(
3	0.02855	6.602620	13.260444	0.603571	1.134681e+33	0.158890	0.749311	0.000(
4	0.02855	5.856012	3.677591	0.890167	1.538030e+33	0.362661	0.755969	0.000(

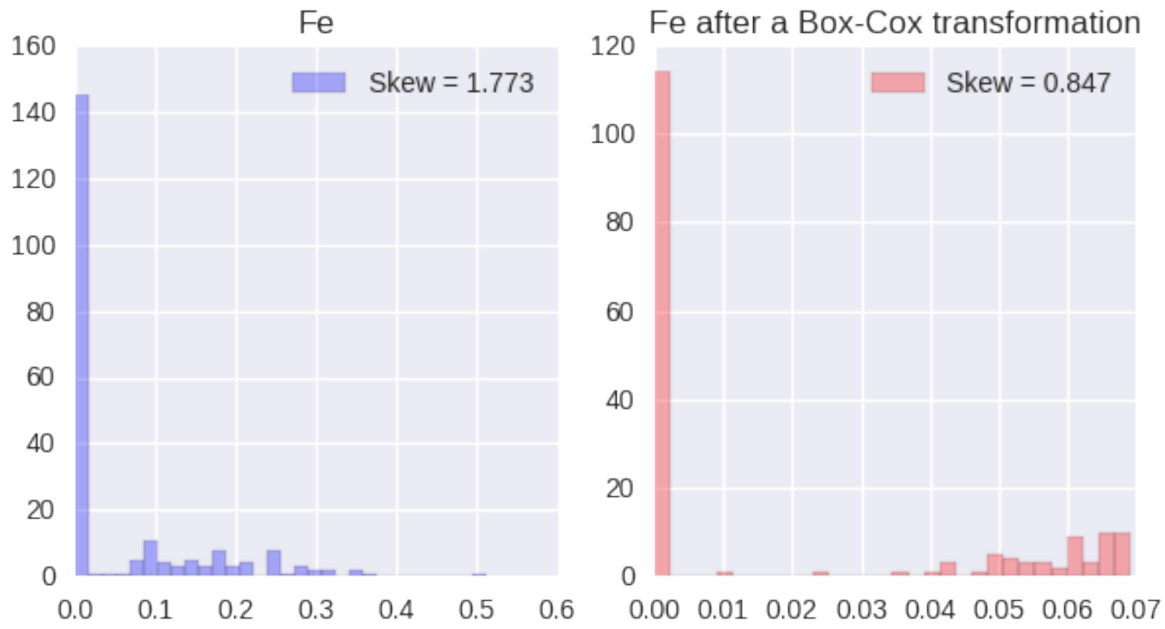
In [15]:
```
for feature in features:
    fig, ax = plt.subplots(1,2,figsize=(7,3.5))
    ax[0].hist(df[feature], color='blue', bins=30, alpha=0.3, label
='Skew = %s' %(str(round(X_train[feature].skew(),3))) )
    ax[0].set_title(str(feature))
    ax[0].legend(loc=0)
    ax[1].hist(df_bc[feature], color='red', bins=30, alpha=0.3, lab
el='Skew = %s' %(str(round(df_bc[feature].skew(),3))) )
    ax[1].set_title(str(feature)+' after a Box-Cox transformation')
    ax[1].legend(loc=0)
    plt.show()
```


RI
Skew = 1.604
RI after a Box-Cox transformation
Skew = 0
1e−14+2.8549682368e−2
Na
Skew = 0.107
Na after a Box-Cox transformation
Skew = 0.012


Mg
Skew = -1.143
Mg after a Box-Cox transformation
Skew = -0.879
Al
Skew = 1.026
Al after a Box-Cox transformation
Skew = -0.009


Si
Skew = -0.954
Si after a Box-Cox transformation
Skew = 0.162
1e33
K
Skew = 6.119
K after a Box-Cox transformation
Skew = -0.062


Ca
Skew = 2.05
Ca after a Box-Cox transformation
Skew = -0.202
Ba
Skew = 3.308
Ba after a Box-Cox transformation
Skew = 1.903



In [16]:

```
# check if skew is closer to zero after a box-cox transform
for feature in features:
    delta = np.abs( df_bc[feature].skew() / df[feature].skew() )
    if delta < 1.0 :
        print('Feature %s is less skewed after a Box-Cox transform'
%(feature))
    else:
        print('Feature %s is more skewed after a Box-Cox transform'
%(feature))
```

```
Feature RI is less skewed after a Box-Cox transform
Feature Na is less skewed after a Box-Cox transform
Feature Mg is less skewed after a Box-Cox transform
Feature Al is less skewed after a Box-Cox transform
Feature Si is less skewed after a Box-Cox transform
Feature K is less skewed after a Box-Cox transform
Feature Ca is less skewed after a Box-Cox transform
Feature Ba is less skewed after a Box-Cox transform
Feature Fe is less skewed after a Box-Cox transform
```

The Box-Cox transform seems to do a good job in reducing the skews of the different distributions of features. Next, we will use the transformed features to feed them into out machine learning models. Only the distribution of Si will not be transformed since such transformation leads to very high values without a big improvement in skewness.

In [17]:
```
df_bc["Si"] = df["Si"]

for feature in features:
    if feature not in ["Si"]:
        X_train[feature], lmbda = boxcox(X_train[feature]+1)  # shi
ft by 1 to avoid computing log of negative values
        X_test[feature] = X_test[feature].apply(lambda x: ((x+1.0)*
*lmbda - 1.0)/lmbda if lmbda !=0 else np.log(x+1) )


X_train, X_test = X_train.values, X_test.values
y_train, y_test = y_train.values, y_test.values
```

```
/opt/conda/lib/python3.5/site-packages/ipykernel/__main__.py:5: Se
ttingWithCopyWarning:
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: http://pandas.pydata.org/pan
das-docs/stable/indexing.html#indexing-view-versus-copy
/opt/conda/lib/python3.5/site-packages/ipykernel/__main__.py:6: Se
ttingWithCopyWarning:
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: http://pandas.pydata.org/pan
das-docs/stable/indexing.html#indexing-view-versus-copy
```

- Standarizing the dataset

Now we have to standarize the different features to bring them to the same scale.

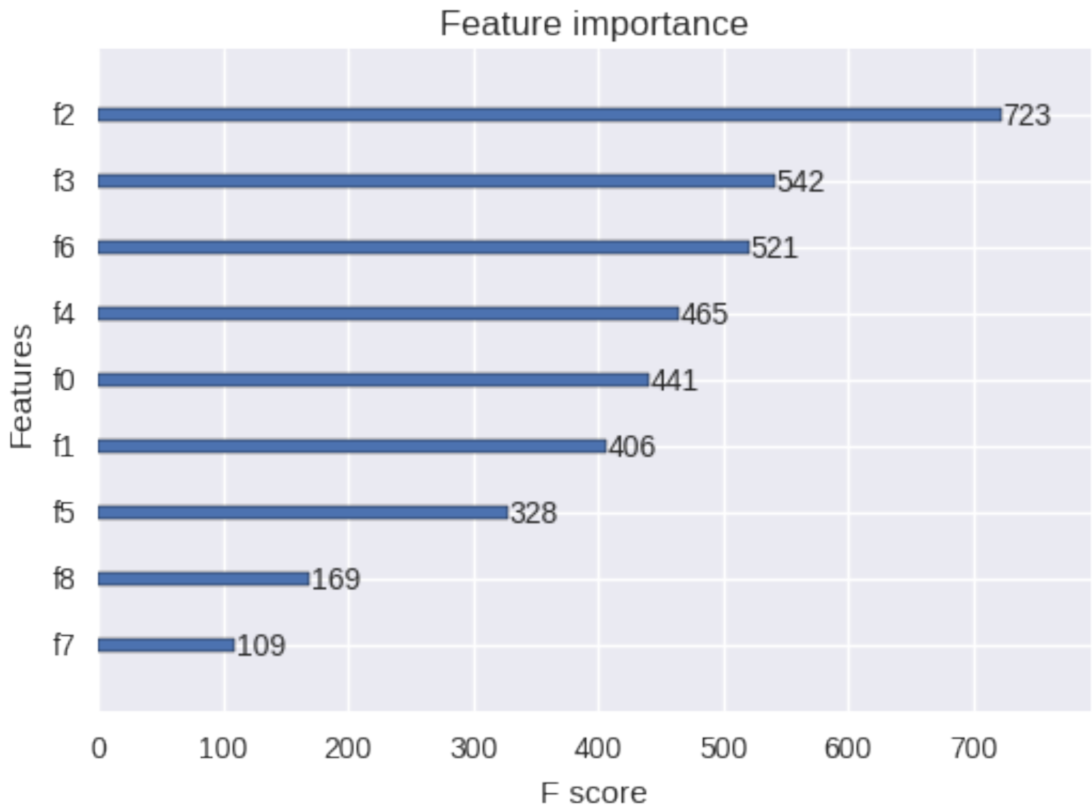
In [18]:
```
# Standarize the dataset
for i in range(X.shape[1]):
    sc = StandardScaler()
    X_train[:,i] = sc.fit_transform(X_train[:,i].reshape(-1,1)).res
hape(1,-1)
    X_test[:,i] = sc.transform(X_test[:,i].reshape(-1,1)).reshape(1
,-1)
```

4. Evaluate Algorithms

- Assessing feature importance via XGBoost and PCA

- **XGBoost**

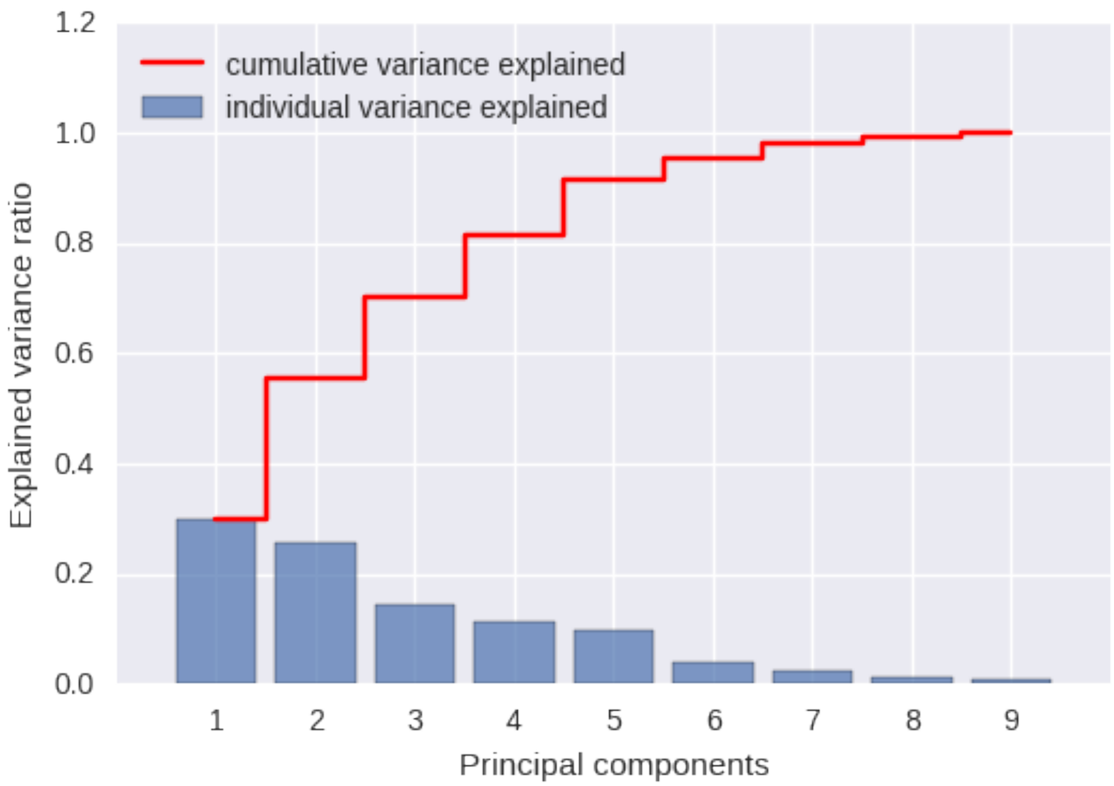
In [19]:
```
model_importances = XGBClassifier(n_estimators=200)
model_importances.fit(X_train, y_train)
plot_importance(model_importances)
plt.show()
```



- **PCA**

Let's go ahead and perform a PCA on the features to decorrelate the ones that are linearly dependent and then let's plot the cumulative explained variance.

In [20]:
```
pca = PCA(random_state = seed)
pca.fit(X_train)
var_exp = pca.explained_variance_ratio_
cum_var_exp = np.cumsum(var_exp)
plt.bar(range(1,len(cum_var_exp)+1), var_exp, align= 'center', labe
l= 'individual variance explained', \
        alpha = 0.7)
plt.step(range(1,len(cum_var_exp)+1), cum_var_exp, where = 'mid' ,
label= 'cumulative variance explained', \
         color= 'red')
plt.ylabel('Explained variance ratio')
plt.xlabel('Principal components')
plt.xticks(np.arange(1,len(var_exp)+1,1))
plt.legend(loc='best')
plt.show()

# Cumulative variance explained
for i, sum in enumerate(cum_var_exp):
    print("PC" + str(i+1), "Cumulative variance: %.3f% %" %(cum_var
_exp[i]*100))
```




```
PC1 Cumulative variance: 30.124%
PC2 Cumulative variance: 55.615%
PC3 Cumulative variance: 70.138%
PC4 Cumulative variance: 81.563%
PC5 Cumulative variance: 91.353%
PC6 Cumulative variance: 95.507%
PC7 Cumulative variance: 97.948%
PC8 Cumulative variance: 99.128%
PC9 Cumulative variance: 100.000%
```

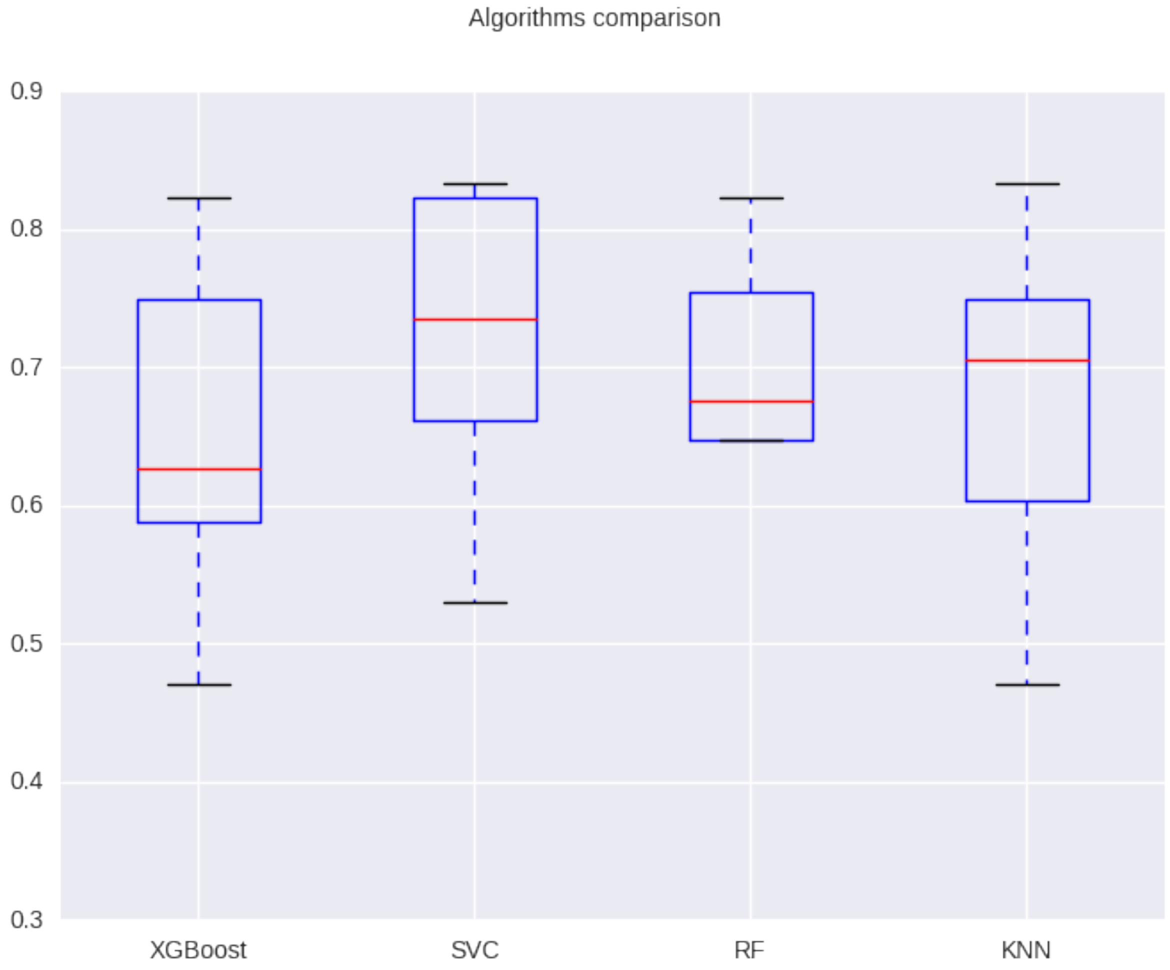
It appears that about 96 % of the variance can be explained with the first 6 principal components. PCA seems a better choice to reduce the dimensionality of the dataset than selecting the most important features via XGBoost.

- Compare Algorithms

Now it's time to compare 4 different algorithms (XGBoost Classifier, Support Vector Classifier, RandomForest Classifier and KNeighbors Classifier) after reducing the dimensionality of the data to 6. We'll use 10-folds cross-validation to assess the performance of each model with the metric being the classification accuracy.

In [21]:
```
pca = PCA(n_components = 6, random_state= seed)
X_train_pca = pca.fit_transform(X_train)
X_test_pca = pca.transform(X_test)

models = []
models.append(('XGBoost', XGBClassifier(seed = seed) ))
models.append(('SVC', SVC(random_state=seed)))
models.append(('RF', RandomForestClassifier(random_state=seed, n_jo
bs=-1 )))
tree = DecisionTreeClassifier(max_depth=4, random_state=seed)
models.append(('KNN', KNeighborsClassifier(n_jobs=-1)))

results, names  = [], []
num_folds = 10
scoring = 'accuracy'

for name, model in models:
    kfold = KFold(n_splits=num_folds, random_state=seed)
    cv_results = cross_val_score(model, X_train_pca, y_train, cv=kf
old, scoring = scoring, n_jobs= -1)
    results.append(cv_results)
    names.append(name)
    msg = "%s: %f (%f)" % (name, cv_results.mean(), cv_results.std(
))
    print(msg)

fig = plt.figure(figsize=(8,6))
fig.suptitle("Algorithms comparison")
ax = fig.add_subplot(1,1,1)
plt.boxplot(results)
ax.set_xticklabels(names)
plt.show()
```

```
XGBoost: 0.654902 (0.116068)
SVC: 0.718627 (0.111958)
RF: 0.654575 (0.136020)
KNN: 0.677451 (0.100790)
```



Observation: It appears that the XGBoost Classifier (XGBClassifer), the Support Vector Classifier (SVC) and the KNeigbors Classifier yield the highest scores. However, these algorithms also yield a wide distribution (10% to 13%). It is worthy to continue our study by tuning these two algorithms.

5. Algorithm tuning

Let's start by tuning the hyperparameters of the XGBoost Classifier.

to be continued ...