DSI Summer Workshops Series

July 19, 2018

Peggy Lindner
Center for Advanced Computing & Data Science (CACDS)
Data Science Institute (DSI)
University of Houston
plindner@uh.edu

Please make sure you have Jupyterhub running and all the required python modules installed. Data for this and other tutorials can be found in the github repsoitory for the Summer 2018 DSI Workshops
https://github.com/peggylind/Materials_Summer2018
(https://github.com/peggylind/Materials_Summer2018)

presenting:

Facies classification using Machine Learning

Brendon Hall, Enthought (https://www.enthought.com/)

This notebook demonstrates how to train a machine learning algorithm to predict facies from well log data. The dataset we will use comes from a class excercise from The University of Kansas on Neural Networks and Fuzzy Systems (http://www.people.ku.edu/~gbohling/EECS833/). This exercise is based on a consortium project to use machine learning techniques to create a reservoir model of the largest gas fields in North America, the Hugoton and Panoma Fields. For more info on the origin of the data, see Bohling and Dubois (2003) (http://www.kgs.ku.edu/PRS/publication/2003/ofr2003-50.pdf) and Dubois et al. (2007) (http://dx.doi.org/10.1016/j.cageo.2006.08.011).

The dataset we will use is log data from nine wells that have been labeled with a facies type based on oberservation of core. We will use this log data to train a support vector machine to classify facies types. Support vector machines (or SVMs) are a type of supervised learning model that can be trained on data to perform classification and regression tasks. The SVM algorithm uses the training data to fit an optimal hyperplane between the different classes (or facies, in our case). We will use the SVM implementation in scikit-learn (http://scikit-learn.org/stable/modules/svm.html).

First we will explore the dataset. We will load the training data from 9 wells, and take a look at what we have to work with. We will plot the data from a couple wells, and create cross plots to look at the variation within the data.

Next we will condition the data set. We will remove the entries that have incomplete data. The data will be scaled to have zero mean and unit variance. We will also split the data into training and test sets.

We will then be ready to train the SVM classifier. We will demonstrate how to use the cross validation set to do model parameter selection.

Finally, once we have a built and tuned the classifier, we can apply the trained model to classify facies in wells which do not already have labels. We will apply the classifier to two wells, but in principle you could apply the classifier to any number

Exploring the dataset

First, we will examine the data set we will use to train the classifier. The training data is contained in the file `facies_vectors.csv`. The dataset consists of 5 wireline log measurements, two indicator variables and a facies label at half foot intervals. In machine learning terminology, each log measurement is a feature vector that maps a set of 'features' (the log measurements) to a class (the facies type). We will use the pandas library to load the data into a dataframe, which provides a convenient data structure to work with well log data.

In [2]:

```
%matplotlib inline
import pandas as pd
import numpy as np
import matplotlib as mpl
import matplotlib.pyplot as plt
import matplotlib.colors as colors
from mpl_toolkits.axes_grid1 import make_axes_locatable

from pandas import set_option
set_option("display.max_rows", 10)
pd.options.mode.chained_assignment = None

filename = 'dataJuly19th/facies_vectors.csv'
training_data = pd.read_csv(filename)
training_data
```

Out[2]:

	Facies	Formation	Well Name	Depth	GR	ILD_log
0	3	A1 SH	SHRIMPLIN	2793.0	77.450	0.664
1	3	A1 SH	SHRIMPLIN	2793.5	78.260	0.661
2	3	A1 SH	SHRIMPLIN	2794.0	79.050	0.658
3	3	A1 SH	SHRIMPLIN	2794.5	86.100	0.655
4	3	A1 SH	SHRIMPLIN	2795.0	74.580	0.647
...	...	...	...	...	...	...
4144	5	C LM	CHURCHMAN BIBLE	3120.5	46.719	0.947
4145	5	C LM	CHURCHMAN BIBLE	3121.0	44.563	0.953
4146	5	C LM	CHURCHMAN BIBLE	3121.5	49.719	0.964
4147	5	C LM	CHURCHMAN BIBLE	3122.0	51.469	0.965
4148	5	C LM	CHURCHMAN BIBLE	3122.5	50.031	0.970

4149 rows × 11 columns

Remove a single well to use as a blind test later.

In [3]:

```
blind = training_data[training_data['Well Name'] == 'SHANKLE']
training_data = training_data[training_data['Well Name'] != 'S
HANKLE']
```

This data is from the Council Grove gas reservoir in Southwest Kansas. The Panoma Council Grove Field is predominantly a carbonate gas reservoir encompassing 2700 square miles in Southwestern Kansas. This dataset is from nine wells (with 4149 examples), consisting of a set of seven predictor variables and a rock facies (class) for each example vector and validation (test) data (830 examples from two wells) having the same seven predictor variables in the feature vector. Facies are based on examination of cores from nine wells taken vertically at half-foot intervals. Predictor variables include five from wireline log measurements and two geologic constraining variables that are derived from geologic knowledge. These are essentially continuous variables sampled at a half-foot sample rate.

The seven predictor variables are:

- Five wire line log curves include gamma ray (http://petrowiki.org/Gamma_ray_logs) (GR), resistivity logging (http://petrowiki.org/Resistivity_and_spontaneous_%28SP%29_logging) (ILD_log10), photoelectric effect (http://www.glossary.oilfield.slb.com/en/Terms/p/photoelectric_effect.aspx) (PE), neutron-density porosity difference and average neutron-density porosity (http://petrowiki.org/Neutron_porosity_logs) (DeltaPHI and PHIND). Note, some wells do not have PE.
- Two geologic constraining variables: nonmarine-marine indicator (NM_M) and relative position (RELPOS)

The nine discrete facies (classes of rocks) are:

1. Nonmarine sandstone
2. Nonmarine coarse siltstone
3. Nonmarine fine siltstone
4. Marine siltstone and shale
5. Mudstone (limestone)
6. Wackestone (limestone)
7. Dolomite
8. Packstone-grainstone (limestone)
9. Phylloid-algal bafflestone (limestone)

These facies aren't discrete, and gradually blend into one another. Some have neighboring facies that are rather close. Mislabeling within these neighboring facies can be expected to occur. The following table lists the facies, their abbreviated labels and their approximate neighbors.

Facies	Label	Adjacent Facies
1	SS	2
2	CSiS	1,3
3	FSiS	2
4	SiSh	5
5	MS	4,6
6	WS	5,7
7	D	6,8
8	PS	6,7,9
9	BS	7,8

Let's clean up this dataset. The 'Well Name' and 'Formation' columns can be turned

In [4]:

```
training_data['Well Name'] = training_data['Well Name'].astype
('category')
training_data['Formation'] = training_data['Formation'].astype
('category')
training_data['Well Name'].unique()
```

Out[4]:

```
[SHRIMPLIN, ALEXANDER D, LUKE G U, KIMZEY A, CROSS
H CATTLE, NOLAN, Recruit F9, NEWBY, CHURCHMAN BIBL
E]
Categories (9, object): [SHRIMPLIN, ALEXANDER D, LU
KE G U, KIMZEY A, ..., NOLAN, Recruit F9, NEWBY, CH
URCHMAN BIBLE]
```

These are the names of the 10 training wells in the Council Grove reservoir. Data has been recruited into pseudo-well 'Recruit F9' to better represent facies 9, the Phylloid-algal bafflestone.

Before we plot the well data, let's define a color map so the facies are represented by consistent color in all the plots in this tutorial. We also create the abbreviated facies labels, and add those to the `facies_vectors` dataframe.

In [5]:

```
# 1=sandstone  2=c_siltstone   3=f_siltstone
# 4=marine_silt_shale 5=mudstone 6=wackestone 7=dolomite
# 8=packstone 9=bafflestone
facies_colors = ['#F4D03F', '#F5B041','#DC7633','#6E2C00',
       '#1B4F72','#2E86C1', '#AED6F1', '#A569BD', '#196F3D']

facies_labels = ['SS', 'CSiS', 'FSiS', 'SiSh', 'MS',
                 'WS', 'D','PS', 'BS']
#facies_color_map is a dictionary that maps facies labels
#to their respective colors
facies_color_map = {}
for ind, label in enumerate(facies_labels):
    facies_color_map[label] = facies_colors[ind]

def label_facies(row, labels):
    return labels[ row['Facies'] -1]

training_data.loc[:,'FaciesLabels'] = training_data.apply(lamb
da row: label_facies(row, facies_labels), axis=1)
training_data.describe()
```

Out[5]:

	Facies	Depth	GR	ILD_log10	
count	3700.000000	3700.000000	3700.000000	3700.000000	370
mean	4.615676	2908.853378	64.873649	0.663053	4.6
std	2.475808	139.010855	30.817166	0.253863	5.1
min	1.000000	2573.500000	10.149000	-0.025949	-21
25%	2.000000	2818.500000	43.778250	0.502000	1.8
50%	4.000000	2939.000000	64.817000	0.645613	4.4
75%	7.000000	3015.125000	80.322500	0.823000	7.6
max	9.000000	3138.000000	361.150000	1.800000	19.

This is a quick view of the statistical distribution of the input variables. Looking at the `count` values, most values have 4149 valid values except for `PE`, which has 3232. In this tutorial we will drop the feature vectors that don't have a valid `PE` entry.

In [6]:

```
PE_mask = training_data['PE'].notnull().values
training_data = training_data[PE_mask]
```

Let's take a look at the data from individual wells in a more familiar log plot form. We will create plots for the five well log variables, as well as a log for facies labels. The plots are based on the those described in Alessandro Amato del Monte's excellent tutorial (https://github.com/seg/tutorials/tree/master/1504_Seismic_petrophysics_1).

In [7]:

```
def make_facies_log_plot(logs, facies_colors):
    #make sure logs are sorted by depth
    logs = logs.sort_values(by='Depth')
    cmap_facies = colors.ListedColormap(
            facies_colors[0:len(facies_colors)], 'indexed')

    ztop=logs.Depth.min(); zbot=logs.Depth.max()

    cluster=np.repeat(np.expand_dims(logs['Facies'].values,1),
 100, 1)

    f, ax = plt.subplots(nrows=1, ncols=6, figsize=(8, 12))
    ax[0].plot(logs.GR, logs.Depth, '-g')
    ax[1].plot(logs.ILD_log10, logs.Depth, '-')
    ax[2].plot(logs.DeltaPHI, logs.Depth, '-', color='0.5')
    ax[3].plot(logs.PHIND, logs.Depth, '-', color='r')
    ax[4].plot(logs.PE, logs.Depth, '-', color='black')
    im=ax[5].imshow(cluster, interpolation='none', aspect='aut
o',
                    cmap=cmap_facies,vmin=1,vmax=9)

    divider = make_axes_locatable(ax[5])
    cax = divider.append_axes("right", size="20%", pad=0.05)
    cbar=plt.colorbar(im, cax=cax)
    cbar.set_label((17*' ').join([' SS ', 'CSiS', 'FSiS',
                                  'SiSh', ' MS ', ' WS ', ' D  '
,
                                  ' PS ', ' BS ']))
    cbar.set_ticks(range(0,1)); cbar.set_ticklabels('')

    for i in range(len(ax)-1):
        ax[i].set_ylim(ztop,zbot)
        ax[i].invert_yaxis()
        ax[i].grid()
        ax[i].locator_params(axis='x', nbins=3)

    ax[0].set_xlabel("GR")
    ax[0].set_xlim(logs.GR.min(),logs.GR.max())
    ax[1].set_xlabel("ILD_log10")
    ax[1].set_xlim(logs.ILD_log10.min(),logs.ILD_log10.max())
    ax[2].set_xlabel("DeltaPHI")
```

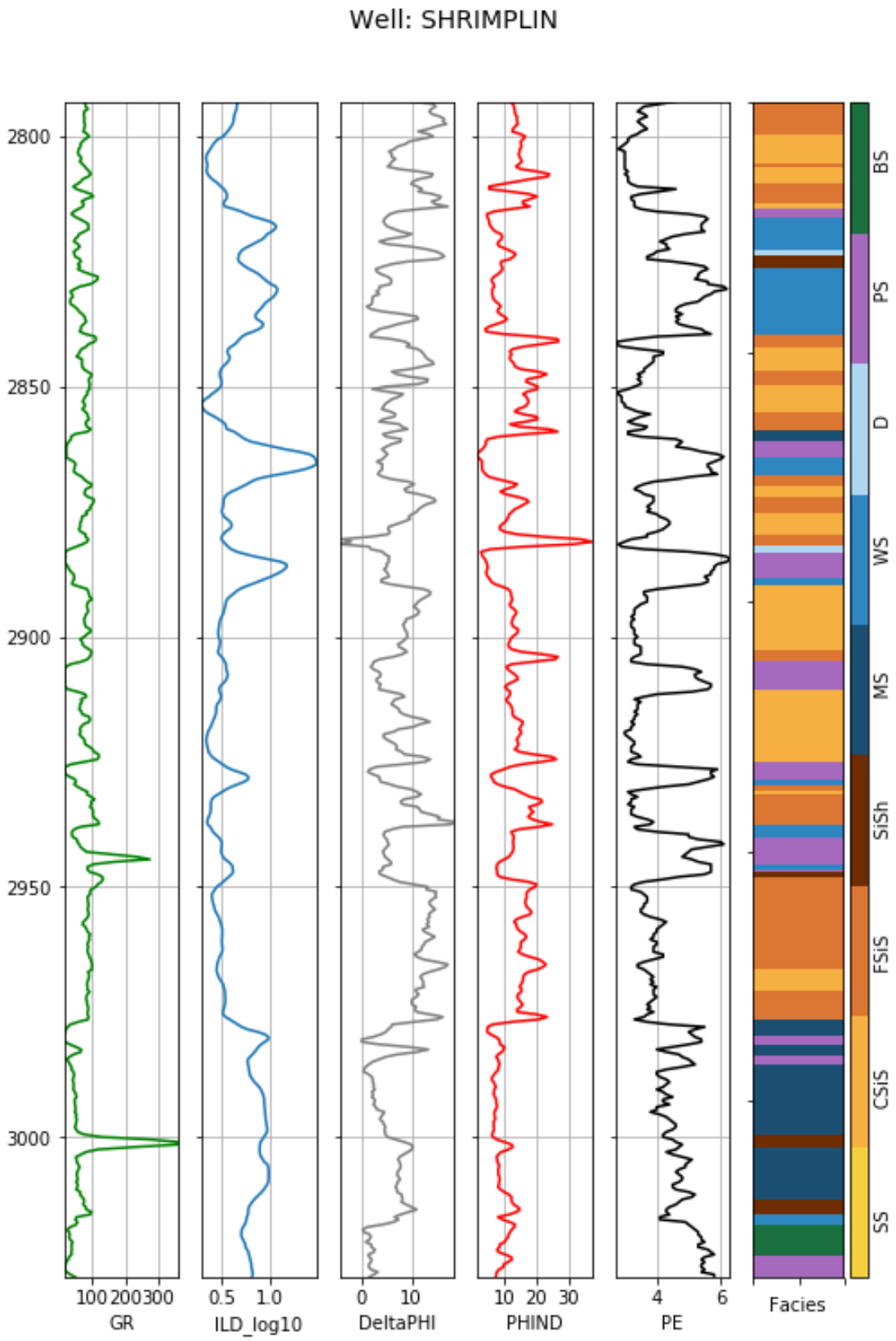
```
    ax[2].set_xlim(logs.DeltaPHI.min(),logs.DeltaPHI.max())
    ax[3].set_xlabel("PHIND")
    ax[3].set_xlim(logs.PHIND.min(),logs.PHIND.max())
    ax[4].set_xlabel("PE")
    ax[4].set_xlim(logs.PE.min(),logs.PE.max())
    ax[5].set_xlabel('Facies')

    ax[1].set_yticklabels([]); ax[2].set_yticklabels([]); ax[3
].set_yticklabels([])
    ax[4].set_yticklabels([]); ax[5].set_yticklabels([])
    ax[5].set_xticklabels([])
    f.suptitle('Well: %s'%logs.iloc[0]['Well Name'], fontsize=
14,y=0.94)
```

Placing the log plotting code in a function will make it easy to plot the logs from multiples wells, and can be reused later to view the results when we apply the facies classification model to other wells. The function was written to take a list of colors and facies labels as parameters.

We then show log plots for wells `SHRIMPLIN`.

In [8]:

```
make_facies_log_plot(
    training_data[training_data['Well Name'] == 'SHRIMPLIN'],
    facies_colors)
```


Well: SHRIMPLIN
2800
2850
2900
2950
3000
100 200 300
GR
0.5 1.0
ILD_log10
0 10
DeltaPHI
10 20 30
PHIND
4 6
PE
Facies
BS
PS
D
WS
MS
SiSh
FSiS
CSiS
SS

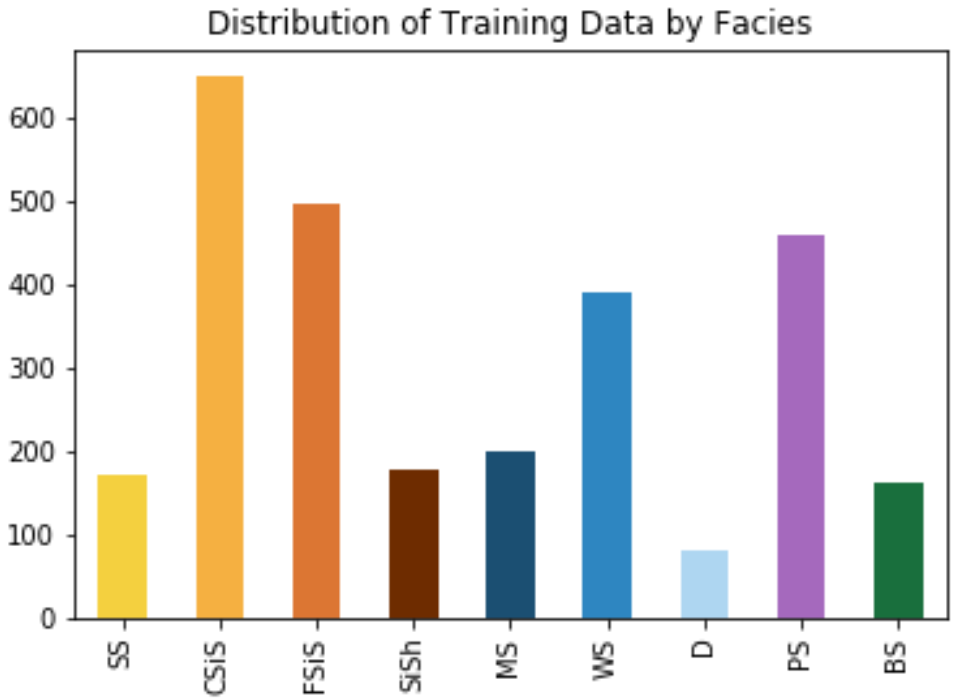
In addition to individual wells, we can look at how the various facies are represented by the entire training set. Let's plot a histgram of the number of training examples for each facies class.

In [9]:

```
#count the number of unique entries for each facies, sort them
 by
#facies number (instead of by number of entries)
facies_counts = training_data['Facies'].value_counts().sort_in
dex()
#use facies labels to index each count
facies_counts.index = facies_labels

facies_counts.plot(kind='bar',color=facies_colors,
                   title='Distribution of Training Data by Fac
ies')
facies_counts
```

Out[9]:

```
SS      170
CSiS    649
FSiS    498
SiSh    177
MS      198
WS      391
D        81
PS      458
BS      161
Name: Facies, dtype: int64
```

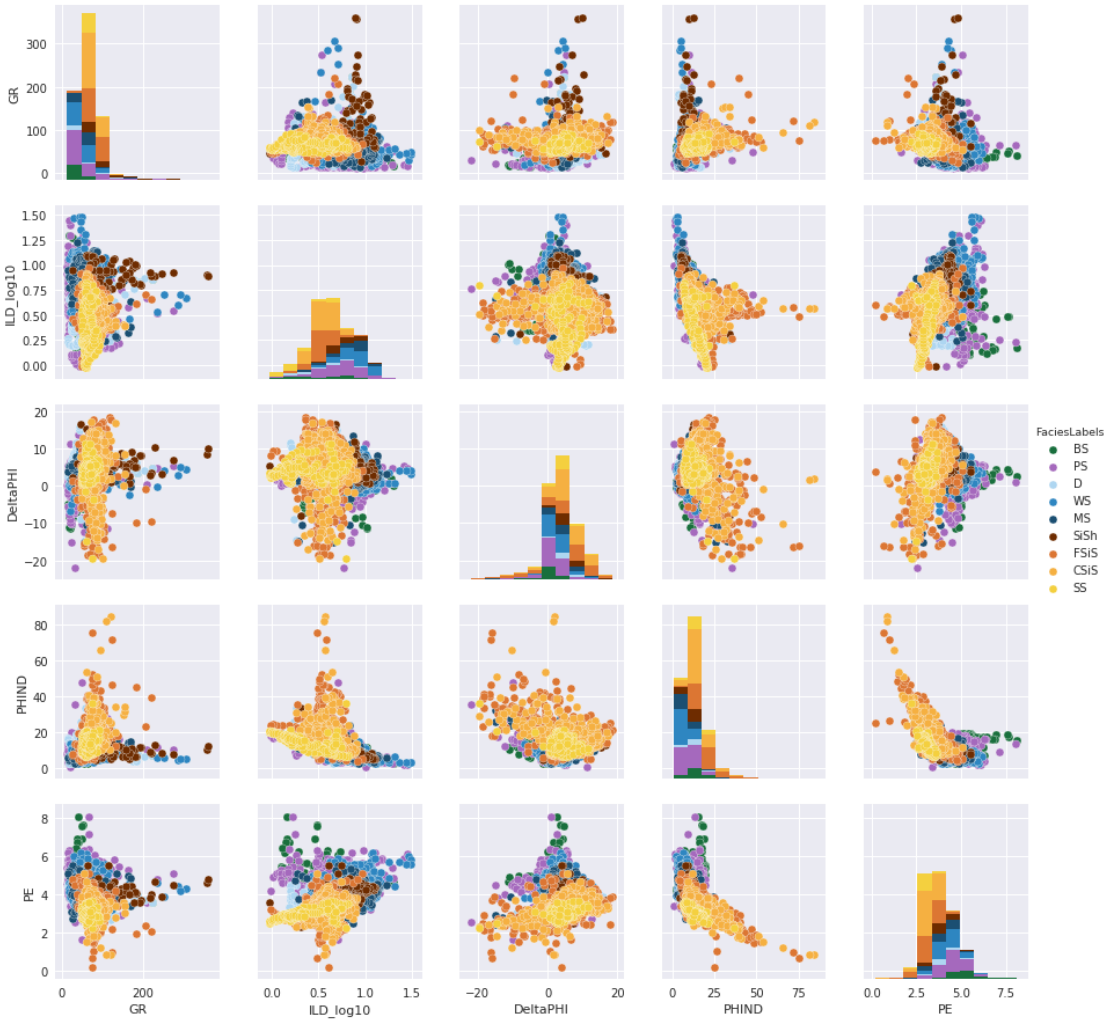


This shows the distribution of examples by facies for the 3232 training examples in the training set. Dolomite (facies 7) has the fewest with 141 examples. There are also only 185 bafflestone examples. Depending on the performance of the classifier we are going to train, we may consider getting more examples of these facies.

Crossplots are a familiar tool in the geosciences to visualize how two properties vary with rock type. This dataset contains 5 log variables, and scatter matrix can help to quickly visualize the variation between the all the variables in the dataset. We can employ the very useful Seaborn library (https://stanford.edu/~mwaskom/software/seaborn/) to quickly create a nice looking scatter matrix. Each pane in the plot shows the relationship between two of the variables on the x and y axis, with each point is colored according to its facies. The same colormap is used to represent the 9 facies.

In [10]:

```
#save plot display settings to change back to when done plotti
ng with seaborn
inline_rc = dict(mpl.rcParams)

import seaborn as sns
sns.set()
sns.pairplot(training_data.drop(['Well Name','Facies','Formati
on','Depth','NM_M','RELPOS'],axis=1),
             hue='FaciesLabels', palette=facies_color_map,
             hue_order=list(reversed(facies_labels)))

#switch back to default matplotlib plot style
mpl.rcParams.update(inline_rc)
```



Conditioning the data set

Now we extract just the feature variables we need to perform the classification. The predictor variables are the five wireline values and two geologic constraining variables. We also get a vector of the facies labels that correspond to each feature vector.

In [11]:

```
correct_facies_labels = training_data['Facies'].values

feature_vectors = training_data.drop(['Formation', 'Well Name'
, 'Depth','Facies','FaciesLabels'], axis=1)
feature_vectors.describe()
```

Out[11]:

	GR	ILD_log10	DeltaPHI	PHIND	
count	2783.000000	2783.000000	2783.000000	2783.000000	278
mean	66.249445	0.644637	3.754963	13.118929	3.8
std	31.610849	0.241922	5.045916	7.389665	0.8
min	13.250000	-0.025949	-21.832000	0.550000	0.2
25%	46.081500	0.497000	1.300000	8.165000	3.2
50%	65.541000	0.627000	3.581000	11.900000	3.6
75%	80.714000	0.812323	6.500000	16.144000	4.4
max	361.150000	1.480000	18.500000	84.400000	8.0

Scikit includes a preprocessing (http://scikit-learn.org/stable/modules/preprocessing.html) module that can 'standardize' the data (giving each variable zero mean and unit variance, also called *whitening*). Many machine learning algorithms assume features will be standard normally distributed data (ie: Gaussian with zero mean and unit variance). The factors used to standardize the training set must be applied to any subsequent feature set that will be input to the classifier. The `StandardScalar` class can be fit to the training set, and later used to standardize any training data.

In [12]:

```
from sklearn import preprocessing

scaler = preprocessing.StandardScaler().fit(feature_vectors)
scaled_features = scaler.transform(feature_vectors)
```

In [13]:

```
feature_vectors
```

Out[13]:

	GR	ILD_log10	DeltaPHI	PHIND	PE	NM_M	RELPC
0	77.450	0.664	9.900	11.915	4.600	1	1.000
1	78.260	0.661	14.200	12.565	4.100	1	0.979
2	79.050	0.658	14.800	13.050	3.600	1	0.957
3	86.100	0.655	13.900	13.115	3.500	1	0.936
4	74.580	0.647	13.500	13.300	3.400	1	0.915
...	...	...	...	...	...	...	...
4144	46.719	0.947	1.828	7.254	3.617	2	0.685
4145	44.563	0.953	2.241	8.013	3.344	2	0.677
4146	49.719	0.964	2.925	8.013	3.190	2	0.669
4147	51.469	0.965	3.083	7.708	3.152	2	0.661
4148	50.031	0.970	2.609	6.668	3.295	2	0.653

2783 rows × 7 columns

Scikit also includes a handy function to randomly split the training data into training and test sets. The test set contains a small subset of feature vectors that are not used to train the network. Because we know the true facies labels for these examples, we can compare the results of the classifier to the actual facies and determine the accuracy of the model. Let's use 20% of the data for the test set.

In [14]:

```
from sklearn.cross_validation import train_test_split

X_train, X_test, y_train, y_test = train_test_split(
        scaled_features, correct_facies_labels, test_size=0.2,
 random_state=42)
```

```
/anaconda3/lib/python3.6/site-packages/sklearn/cros
s_validation.py:41: DeprecationWarning: This module
was deprecated in version 0.18 in favor of the mode
l_selection module into which all the refactored cl
asses and functions are moved. Also note that the i
nterface of the new CV iterators are different from
that of this module. This module will be removed in
0.20.
  "This module will be removed in 0.20.", Deprecati
onWarning)
```

Training the SVM classifier

Now we use the cleaned and conditioned training set to create a facies classifier. As mentioned above, we will use a type of machine learning model known as a support vector machine (https://en.wikipedia.org/wiki/Support_vector_machine). The SVM is a map of the feature vectors as points in a multi dimensional space, mapped so that examples from different facies are divided by a clear gap that is as wide as possible.

The SVM implementation in scikit-learn (http://scikit-learn.org/stable/modules/generated/sklearn.svm.SVC.html#sklearn.svm.SVC) takes a number of important parameters. First we create a classifier using the default settings.

In [15]:

```
from sklearn import svm

clf = svm.SVC()
```

Now we can train the classifier using the training set we created above.

In [16]:

```
clf.fit(X_train,y_train)
```

Out[16]:

```
SVC(C=1.0, cache_size=200, class_weight=None, coef0
=0.0,
  decision_function_shape='ovr', degree=3, gamma='a
uto', kernel='rbf',
  max_iter=-1, probability=False, random_state=Non
e, shrinking=True,
  tol=0.001, verbose=False)
```

Now that the model has been trained on our data, we can use it to predict the facies of the feature vectors in the test set. Because we know the true facies labels of the vectors in the test set, we can use the results to evaluate the accuracy of the classifier.

In [19]:

```
predicted_labels = clf.predict(X_test)
```

We need some metrics to evaluate how good our classifier is doing. A confusion matrix (http://www.dataschool.io/simple-guide-to-confusion-matrix-terminology/) is a table that can be used to describe the performance of a classification model. Scikit-learn (http://scikit-learn.org/stable/modules/generated/sklearn.metrics.confusion_matrix.html) allows us to easily create a confusion matrix by supplying the actual and predicted facies labels.

The confusion matrix is simply a 2D array. The entries of confusion matrix `C[i][j]` are equal to the number of observations predicted to have facies `j`, but are known to have facies `i`.

To simplify reading the confusion matrix, a function has been written to display the matrix along with facies labels and various error metrics. See the file `classification_utilities.py` in this repo for the `display_cm()` function.

In [20]:

```
from sklearn.metrics import confusion_matrix
from classification_utilities import display_cm, display_adj_cm

conf = confusion_matrix(y_test, predicted_labels)
display_cm(conf, facies_labels, hide_zeros=True)
```

```
     Pred    SS  CSiS  FSiS  SiSh    MS    WS     D
    PS    BS Total
     True
       SS    19    19     1
                  39
     CSiS          102    28
                 130
     FSiS           33    49           1
     1             84
     SiSh            1          24           6
     2             33
       MS            2     1     2     2    42     1
    10             60
       WS                        7          43     2
    19     1    72
        D                        1                 9
     7             17
       PS                  2     1     2    22
    56     5    88
       BS                                  1
    11    22    34
```

The rows of the confusion matrix correspond to the actual facies labels. The columns correspond to the labels assigned by the classifier. For example, consider the first row. For the feature vectors in the test set that actually have label `SS`, 23 were correctly indentified as `SS`, 21 were classified as `CSiS` and 2 were classified as `FSiS`.

The entries along the diagonal are the facies that have been correctly classified. Below we define two functions that will give an overall value for how the algorithm is performing. The accuracy is defined as the number of correct classifications divided by the total number of classifications.

In [21]:

```
def accuracy(conf):
    total_correct = 0.
    nb_classes = conf.shape[0]
    for i in np.arange(0,nb_classes):
        total_correct += conf[i][i]
    acc = total_correct/sum(sum(conf))
    return acc
```

As noted above, the boundaries between the facies classes are not all sharp, and some of them blend into one another. The error within these 'adjacent facies' can also be calculated. We define an array to represent the facies adjacent to each other. For facies label `i`, `adjacent_facies[i]` is an array of the adjacent facies labels.

In [22]:

```
adjacent_facies = np.array([[1], [0,2], [1], [4], [3,5], [4,6,
7], [5,7], [5,6,8], [6,7]])

def accuracy_adjacent(conf, adjacent_facies):
    nb_classes = conf.shape[0]
    total_correct = 0.
    for i in np.arange(0,nb_classes):
        total_correct += conf[i][i]
        for j in adjacent_facies[i]:
            total_correct += conf[i][j]
    return total_correct / sum(sum(conf))
```

In [23]:

```
print('Facies classification accuracy = %f' % accuracy(conf))
print('Adjacent facies classification accuracy = %f' % accurac
y_adjacent(conf, adjacent_facies))
```

```
Facies classification accuracy = 0.585278
Adjacent facies classification accuracy = 0.926391
```

Model parameter selection

The classifier so far has been built with the default parameters. However, we may be able to get improved classification results with optimal parameter choices.

We will consider two parameters. The parameter `C` is a regularization factor, and tells the classifier how much we want to avoid misclassifying training examples. A large value of C will try to correctly classify more examples from the training set, but if `C` is too large it may 'overfit' the data and fail to generalize when classifying new data. If `C` is too small then the model will not be good at fitting outliers and will have a large error on the training set.

The SVM learning algorithm uses a kernel function to compute the distance between feature vectors. Many kernel functions exist, but in this case we are using the radial basis function `rbf` kernel (the default). The `gamma` parameter describes the size of the radial basis functions, which is how far away two vectors in the feature space need to be to be considered close.

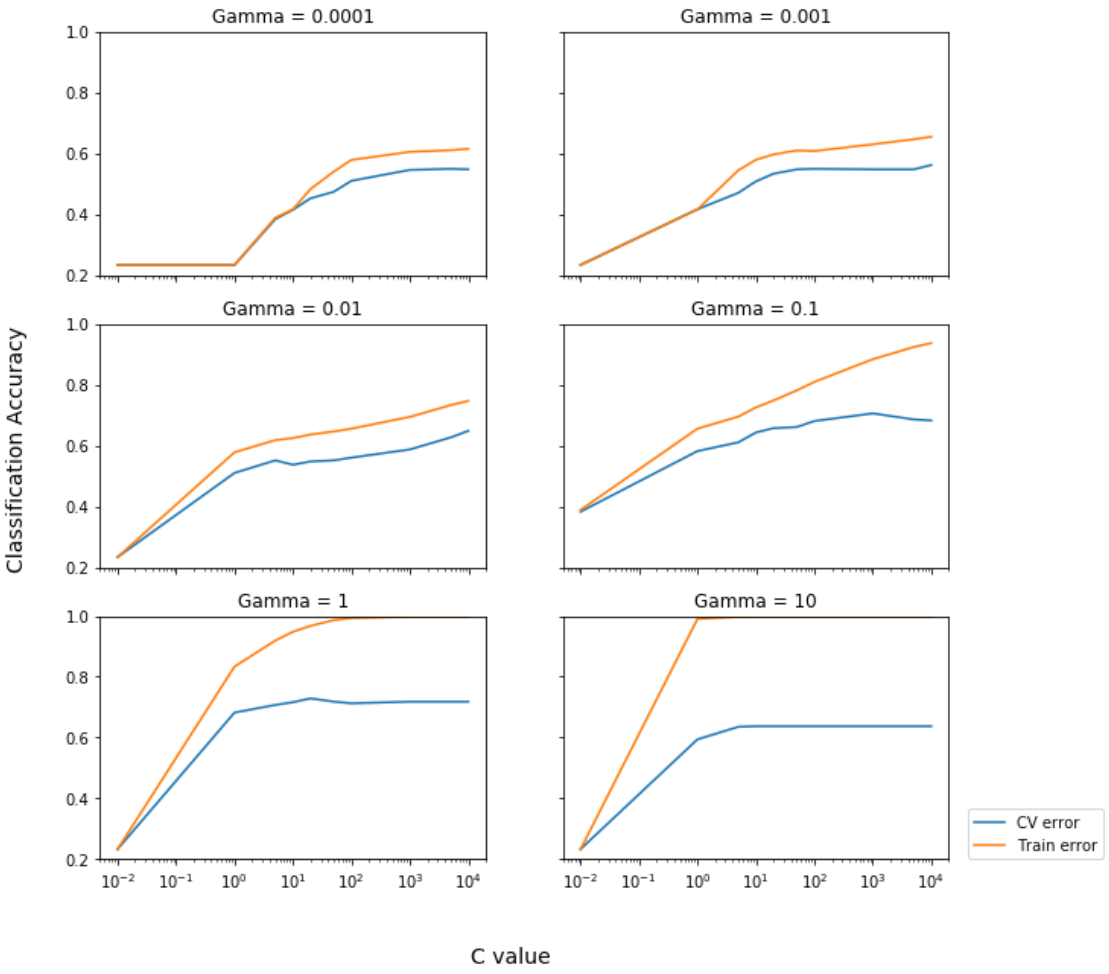
We will train a series of classifiers with different values for `C` and `gamma`. Two nested loops are used to train a classifier for every possible combination of values in the ranges specified. The classification accuracy is recorded for each combination of parameter values. The results are shown in a series of plots, so the parameter values that give the best classification accuracy on the test set can be selected.

This process is also known as 'cross validation'. Often a separate 'cross validation' dataset will be created in addition to the training and test sets to do model selection. For this tutorial we will just use the test set to choose model parameters.

In [24]:

```
#model selection takes a few minutes, change this variable
#to true to run the parameter loop
do_model_selection = True

if do_model_selection:
    C_range = np.array([.01, 1, 5, 10, 20, 50, 100, 1000, 5000
, 10000])
    gamma_range = np.array([0.0001, 0.001, 0.01, 0.1, 1, 10])

    fig, axes = plt.subplots(3, 2,
                         sharex='col', sharey='row',figsize=(10
,10))
    plot_number = 0
    for outer_ind, gamma_value in enumerate(gamma_range):
        row = int(plot_number / 2)
        column = int(plot_number % 2)
        cv_errors = np.zeros(C_range.shape)
        train_errors = np.zeros(C_range.shape)
        for index, c_value in enumerate(C_range):

            clf = svm.SVC(C=c_value, gamma=gamma_value)
            clf.fit(X_train,y_train)

            train_conf = confusion_matrix(y_train, clf.predict
(X_train))
            cv_conf = confusion_matrix(y_test, clf.predict(X_t
est))

            cv_errors[index] = accuracy(cv_conf)
            train_errors[index] = accuracy(train_conf)

        ax = axes[row, column]
        ax.set_title('Gamma = %g'%gamma_value)
        ax.semilogx(C_range, cv_errors, label='CV error')
        ax.semilogx(C_range, train_errors, label='Train error'
)
        plot_number += 1
        ax.set_ylim([0.2,1])

    ax.legend(bbox_to_anchor=(1.05, 0), loc='lower left', bord
eraxespad=0.)
```

```
    fig.text(0.5, 0.03, 'C value', ha='center',
             fontsize=14)

    fig.text(0.04, 0.5, 'Classification Accuracy', va='center'
,
             rotation='vertical', fontsize=14)
```



The best accuracy on the cross validation error curve was achieved for `gamma = 1`, and `C = 10`. We can now create and train an optimized classifier based on these parameters:

In [25]:

```
clf = svm.SVC(C=10, gamma=1)
clf.fit(X_train, y_train)

cv_conf = confusion_matrix(y_test, clf.predict(X_test))

print('Optimized facies classification accuracy = %.2f' % accu
racy(cv_conf))
print('Optimized adjacent facies classification accuracy = %.2
f' % accuracy_adjacent(cv_conf, adjacent_facies))
```

```
Optimized facies classification accuracy = 0.72
Optimized adjacent facies classification accuracy =
0.92
```

Precision and recall (https://en.wikipedia.org/wiki/Precision_and_recall) are metrics that give more insight into how the classifier performs for individual facies. Precision is the probability that given a classification result for a sample, the sample actually belongs to that class. Recall is the probability that a sample will be correctly classified for a given class.

Precision and recall can be computed easily using the confusion matrix. The code to do so has been added to the `display_confusion_matrix()` function:

In [26]:

```
display_cm(cv_conf, facies_labels,
           display_metrics=True, hide_zeros=True)
```

```
     Pred    SS  CSiS  FSiS  SiSh    MS    WS     D
    PS    BS Total
     True
       SS    30     8     1
                39
     CSiS     2   109    17
     2         130
     FSiS     2    23    55     3     1
                84
     SiSh           1          27     1     2
     2          33
       MS           2     2     5    29    14
     8          60
       WS                      7     8    45     1
     9     2    72
        D                      1     2          10
     3     1    17
       PS                3     1     3    14
    63     4    88
       BS
     3    31    34

Precision  0.88  0.76  0.71  0.61  0.66  0.60  0.91
  0.70  0.82  0.72
   Recall  0.77  0.84  0.65  0.82  0.48  0.62  0.59
  0.72  0.91  0.72
       F1  0.82  0.80  0.68  0.70  0.56  0.61  0.71
  0.71  0.86  0.71
```

To interpret these results, consider facies SS. In our test set, if a sample was labeled SS the probability the sample was correct is 0.8 (precision). If we know a sample has facies SS, then the probability it will be correctly labeled by the classifier is 0.78 (recall). It is desirable to have high values for both precision and recall, but often when an algorithm is tuned to increase one, the other decreases. The F1 score (https://en.wikipedia.org/wiki/Precision_and_recall#F-measure) combines both to give a single measure of relevancy of the classifier results.

These results can help guide intuition for how to improve the classifier results. For example, for a sample with facies MS or mudstone, it is only classified correctly 57% of the time (recall). Perhaps this could be improved by introducing more training samples. Sample quality could also play a role. Facies BS or bafflestone has the best F1 score and relatively few training examples. But this data was handpicked from other wells to provide training examples to identify this facies.

We can also consider the classification metrics when we consider misclassifying an adjacent facies as correct:

In [27]:

```
display_adj_cm(cv_conf, facies_labels, adjacent_facies,
           display_metrics=True, hide_zeros=True)
```

```
     Pred    SS  CSiS  FSiS  SiSh    MS    WS     D
    PS    BS Total
     True
       SS    38           1
                39
     CSiS         128
     2         130
     FSiS     2          78     3     1
                84
     SiSh           1          28           2
     2          33
       MS           2     2          48
     8          60
       WS                      7          63
           2    72
        D                      1     2          13
           1    17
       PS                  3     1     3
    81          88
       BS
          34    34

Precision  0.95  0.98  0.93  0.70  0.89  0.97  1.00
  0.87  0.92  0.92
   Recall  0.97  0.98  0.93  0.85  0.80  0.88  0.76
  0.92  1.00  0.92
       F1  0.96  0.98  0.93  0.77  0.84  0.92  0.87
  0.90  0.96  0.92
```

Considering adjacent facies, the `F1` scores for all facies types are above 0.9, except when classifying `SiSh` or marine siltstone and shale. The classifier often misclassifies this facies (recall of 0.66), most often as wackestone.

These results are comparable to those reported in Dubois et al. (2007).

Applying the classification model to the blind data

We held a well back from the training, and stored it in a dataframe called `blind`:

In [28]:

```
blind
```

Out[28]:

	Facies	Formation	Well Name	Depth	GR	ILD_log10	D
937	2	A1 SH	SHANKLE	2774.5	98.36	0.642	-0
938	2	A1 SH	SHANKLE	2775.0	97.57	0.631	7.
939	2	A1 SH	SHANKLE	2775.5	98.41	0.615	12
940	2	A1 SH	SHANKLE	2776.0	85.92	0.597	13
941	2	A1 SH	SHANKLE	2776.5	83.16	0.592	12
...	...	...	...	...	...	...	...
1381	8	C LM	SHANKLE	3006.0	32.84	1.120	-2
1382	8	C LM	SHANKLE	3006.5	25.16	1.112	-1
1383	8	C LM	SHANKLE	3007.0	25.16	1.112	-1
1384	4	C LM	SHANKLE	3007.5	26.22	1.092	-0
1385	4	C LM	SHANKLE	3008.0	65.36	1.026	1.

449 rows × 11 columns

The label vector is just the `Facies` column:

In [29]:

```
y_blind = blind['Facies'].values
```

We can form the feature matrix by dropping some of the columns and making a new dataframe:

In [30]:

```
well_features = blind.drop(['Facies', 'Formation', 'Well Name'
, 'Depth'], axis=1)
```

Now we can transform this with the scaler we made before:

In [31]:

```
X_blind = scaler.transform(well_features)
```

Now it's a simple matter of making a prediction and storing it back in the dataframe:

In [32]:

```
y_pred = clf.predict(X_blind)
blind['Prediction'] = y_pred
```

Let's see how we did with the confusion matrix:

In [33]:

```
cv_conf = confusion_matrix(y_blind, y_pred)

print('Optimized facies classification accuracy = %.2f' % accu
racy(cv_conf))
print('Optimized adjacent facies classification accuracy = %.2
f' % accuracy_adjacent(cv_conf, adjacent_facies))
```

```
Optimized facies classification accuracy = 0.41
Optimized adjacent facies classification accuracy =
0.87
```

We managed 0.75 using the test data, but it was from the same wells as the training data. This more reasonable test does not perform as well...

In [34]:

```
display_cm(cv_conf, facies_labels,
           display_metrics=True, hide_zeros=True)
```

```
     Pred    SS  CSiS  FSiS  SiSh    MS    WS     D
    PS    BS Total
     True
       SS     6    60    23
                 89
     CSiS    14    61    14
                 89
     FSiS     2    51    63
     1          117
     SiSh                      1           5
     1            7
       MS                      3     1     4
     9     2    19
       WS                  3     3    21    28
    16           71
        D                  4     1     3          2
     7           17
       PS                          1    12     2
    24     1    40
       BS
                 0

Precision  0.27  0.35  0.59  0.12  0.04  0.57  0.50
  0.41  0.00  0.43
   Recall  0.07  0.69  0.54  0.14  0.05  0.39  0.12
  0.60  0.00  0.41
       F1  0.11  0.47  0.56  0.13  0.04  0.47  0.19
  0.49  0.00  0.39
```

...but does remarkably well on the adjacent facies predictions.

In [35]:

```
display_adj_cm(cv_conf, facies_labels, adjacent_facies,
               display_metrics=True, hide_zeros=True)
```

```
     Pred    SS  CSiS  FSiS  SiSh    MS    WS     D
    PS    BS Total
     True
       SS    66           23
                89
     CSiS           89
                89
     FSiS     2          114
     1          117
     SiSh                        1           5
     1            7
       MS                              8
     9     2    19
       WS                  3     3          65
                71
        D                  4     1     3           9
                17
       PS                              1
    39           40
       BS
                 0

Precision  0.97  1.00  0.79  0.20  0.67  0.93  1.00
  0.78  0.00  0.88
   Recall  0.74  1.00  0.97  0.14  0.42  0.92  0.53
  0.97  0.00  0.87
       F1  0.84  1.00  0.87  0.17  0.52  0.92  0.69
  0.87  0.00  0.87
```

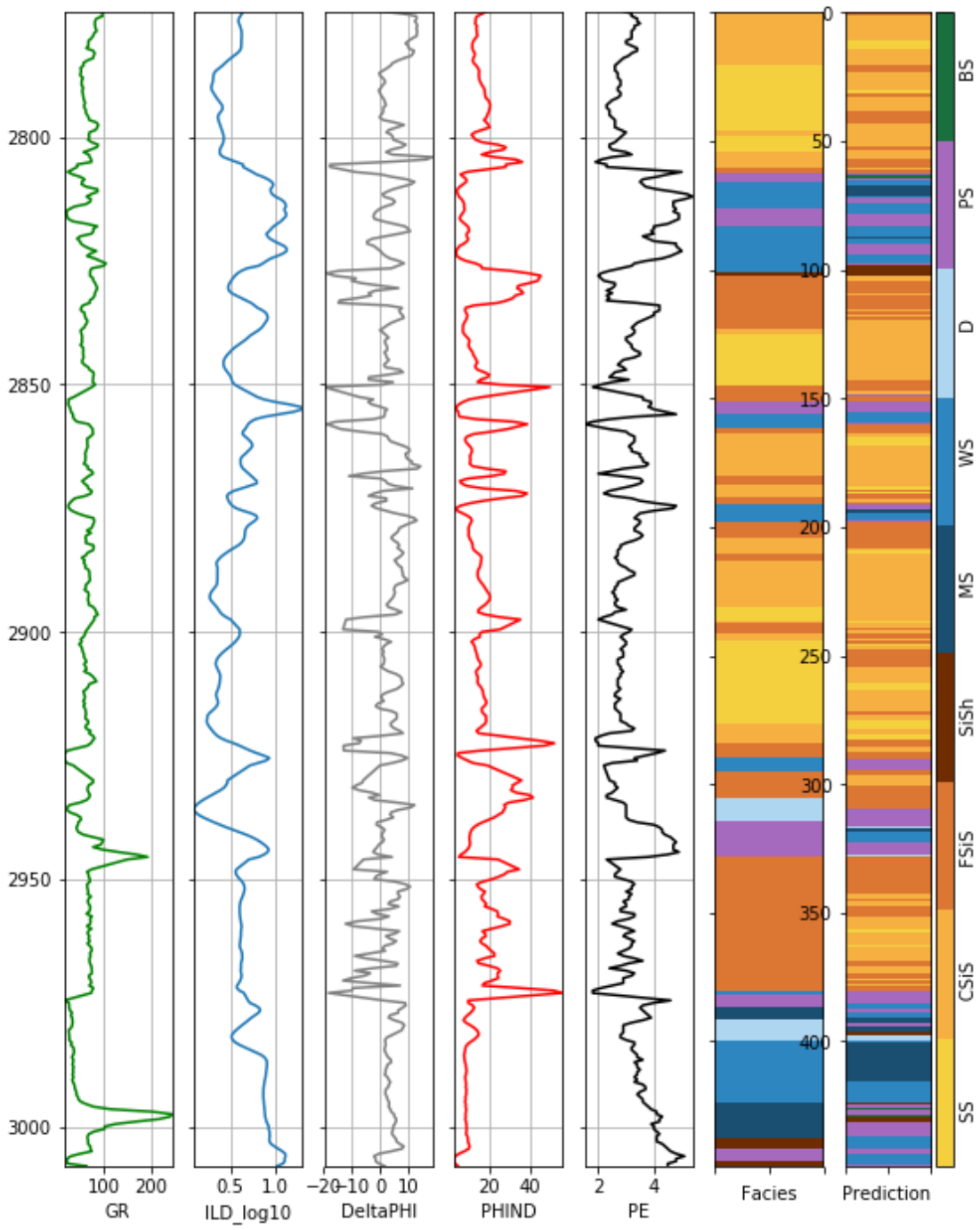
In [36]:

```
def compare_facies_plot(logs, compadre, facies_colors):
    #make sure logs are sorted by depth
    logs = logs.sort_values(by='Depth')
    cmap_facies = colors.ListedColormap(
            facies_colors[0:len(facies_colors)], 'indexed')

    ztop=logs.Depth.min(); zbot=logs.Depth.max()

    cluster1 = np.repeat(np.expand_dims(logs['Facies'].values,
1), 100, 1)
    cluster2 = np.repeat(np.expand_dims(logs[compadre].values,
1), 100, 1)

    f, ax = plt.subplots(nrows=1, ncols=7, figsize=(9, 12))
    ax[0].plot(logs.GR, logs.Depth, '-g')
    ax[1].plot(logs.ILD_log10, logs.Depth, '-')
    ax[2].plot(logs.DeltaPHI, logs.Depth, '-', color='0.5')
    ax[3].plot(logs.PHIND, logs.Depth, '-', color='r')
    ax[4].plot(logs.PE, logs.Depth, '-', color='black')
    im1 = ax[5].imshow(cluster1, interpolation='none', aspect=
'auto',
                       cmap=cmap_facies,vmin=1,vmax=9)
    im2 = ax[6].imshow(cluster2, interpolation='none', aspect=
'auto',
                       cmap=cmap_facies,vmin=1,vmax=9)

    divider = make_axes_locatable(ax[6])
    cax = divider.append_axes("right", size="20%", pad=0.05)
    cbar=plt.colorbar(im2, cax=cax)
    cbar.set_label((17*' ').join([' SS ', 'CSiS', 'FSiS',
                                  'SiSh', ' MS ', ' WS ', ' D  '
,
                                  ' PS ', ' BS ']))
    cbar.set_ticks(range(0,1)); cbar.set_ticklabels('')

    for i in range(len(ax)-2):
        ax[i].set_ylim(ztop,zbot)
        ax[i].invert_yaxis()
        ax[i].grid()
        ax[i].locator_params(axis='x', nbins=3)
```

```
    ax[0].set_xlabel("GR")
    ax[0].set_xlim(logs.GR.min(),logs.GR.max())
    ax[1].set_xlabel("ILD_log10")
    ax[1].set_xlim(logs.ILD_log10.min(),logs.ILD_log10.max())
    ax[2].set_xlabel("DeltaPHI")
    ax[2].set_xlim(logs.DeltaPHI.min(),logs.DeltaPHI.max())
    ax[3].set_xlabel("PHIND")
    ax[3].set_xlim(logs.PHIND.min(),logs.PHIND.max())
    ax[4].set_xlabel("PE")
    ax[4].set_xlim(logs.PE.min(),logs.PE.max())
    ax[5].set_xlabel('Facies')
    ax[6].set_xlabel(compadre)

    ax[1].set_yticklabels([]); ax[2].set_yticklabels([]); ax[3
].set_yticklabels([])
    ax[4].set_yticklabels([]); ax[5].set_yticklabels([])
    ax[5].set_xticklabels([])
    ax[6].set_xticklabels([])
    f.suptitle('Well: %s'%logs.iloc[0]['Well Name'], fontsize=
14,y=0.94)
```

In [37]:

```
compare_facies_plot(blind, 'Prediction', facies_colors)
```


Well: SHANKLE
2800
2850
2900
2950
3000
100 200
GR
0.5 1.0
ILD_log10
−20−10 0 10
DeltaPHI
20 40
PHIND
2 4
PE
0
50
100
150
200
250
300
350
400
Facies
Prediction
BS
PS
D
WS
MS
SiSh
FSiS
CSiS
SS

Applying the classification model to new data

Now that we have a trained facies classification model we can use it to identify facies in wells that do not have core data. In this case, we will apply the classifier to two wells, but we could use it on any number of wells for which we have the same set of well logs for input.

This dataset is similar to the training data except it does not have facies labels. It is loaded into a dataframe called `test_data`.

In [39]:

```
well_data = pd.read_csv('dataJuly19th/validation_data_nofacies.csv')
well_data['Well Name'] = well_data['Well Name'].astype('category')
well_features = well_data.drop(['Formation', 'Well Name', 'Depth'], axis=1)
```

The data needs to be scaled using the same constants we used for the training data.

In [40]:

```
X_unknown = scaler.transform(well_features)
```

Finally we predict facies labels for the unknown data, and store the results in a `Facies` column of the `test_data` dataframe.

In [41]:

```
#predict facies of unclassified data
y_unknown = clf.predict(X_unknown)
well_data['Facies'] = y_unknown
well_data
```

Out[41]:

	Formation	Well Name	Depth	GR	ILD_log10	DeltaPH
0	A1 SH	STUART	2808.0	66.276	0.630	3.300
1	A1 SH	STUART	2808.5	77.252	0.585	6.500
2	A1 SH	STUART	2809.0	82.899	0.566	9.400
3	A1 SH	STUART	2809.5	80.671	0.593	9.500
4	A1 SH	STUART	2810.0	75.971	0.638	8.700
...	...	...	...	...	...	...
825	C SH	CRAWFORD	3158.5	86.078	0.554	5.040
826	C SH	CRAWFORD	3159.0	88.855	0.539	5.560
827	C SH	CRAWFORD	3159.5	90.490	0.530	6.360
828	C SH	CRAWFORD	3160.0	90.975	0.522	7.035
829	C SH	CRAWFORD	3160.5	90.108	0.513	7.505

830 rows × 11 columns

In [42]:

```
well_data['Well Name'].unique()
```

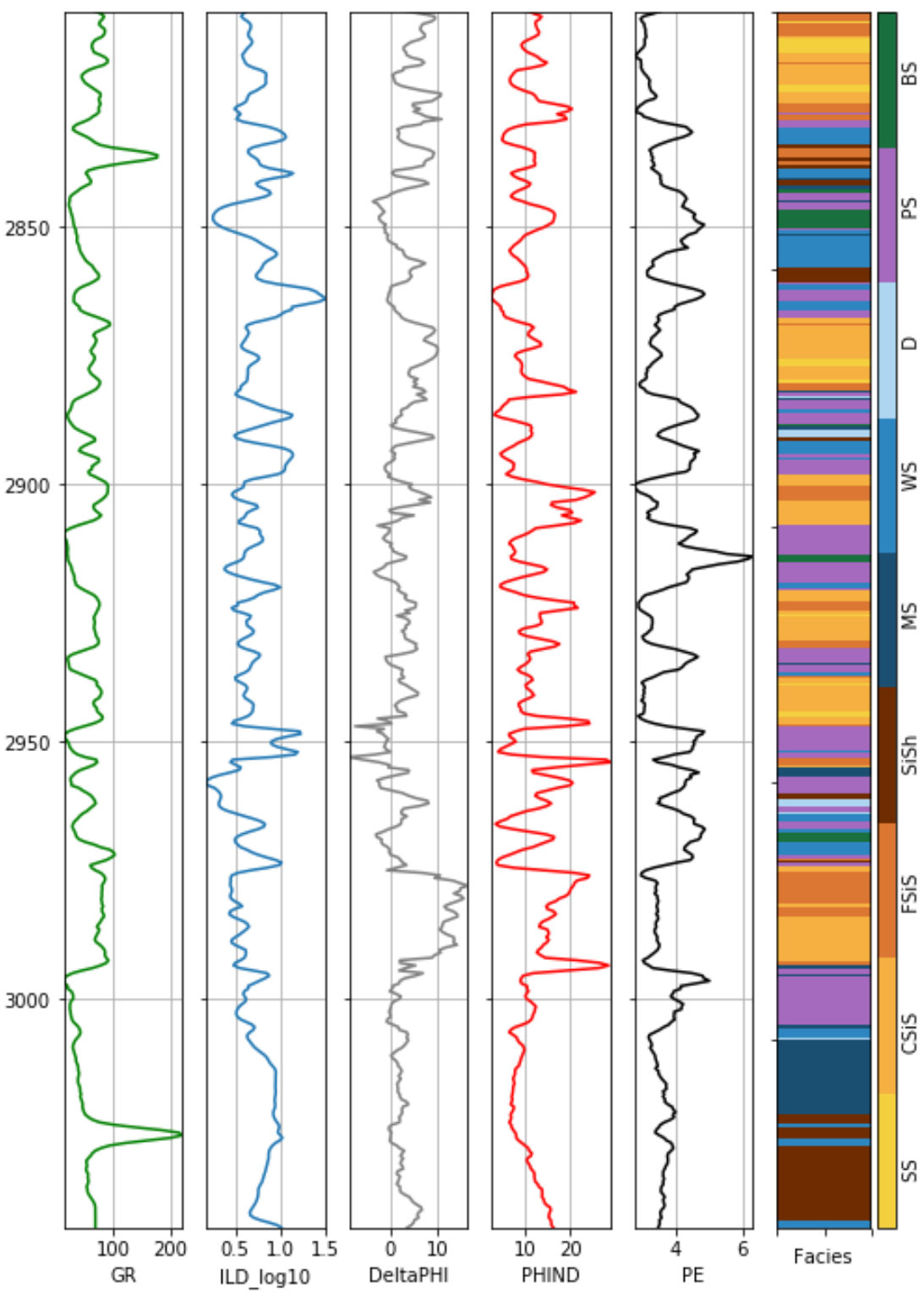
Out[42]:

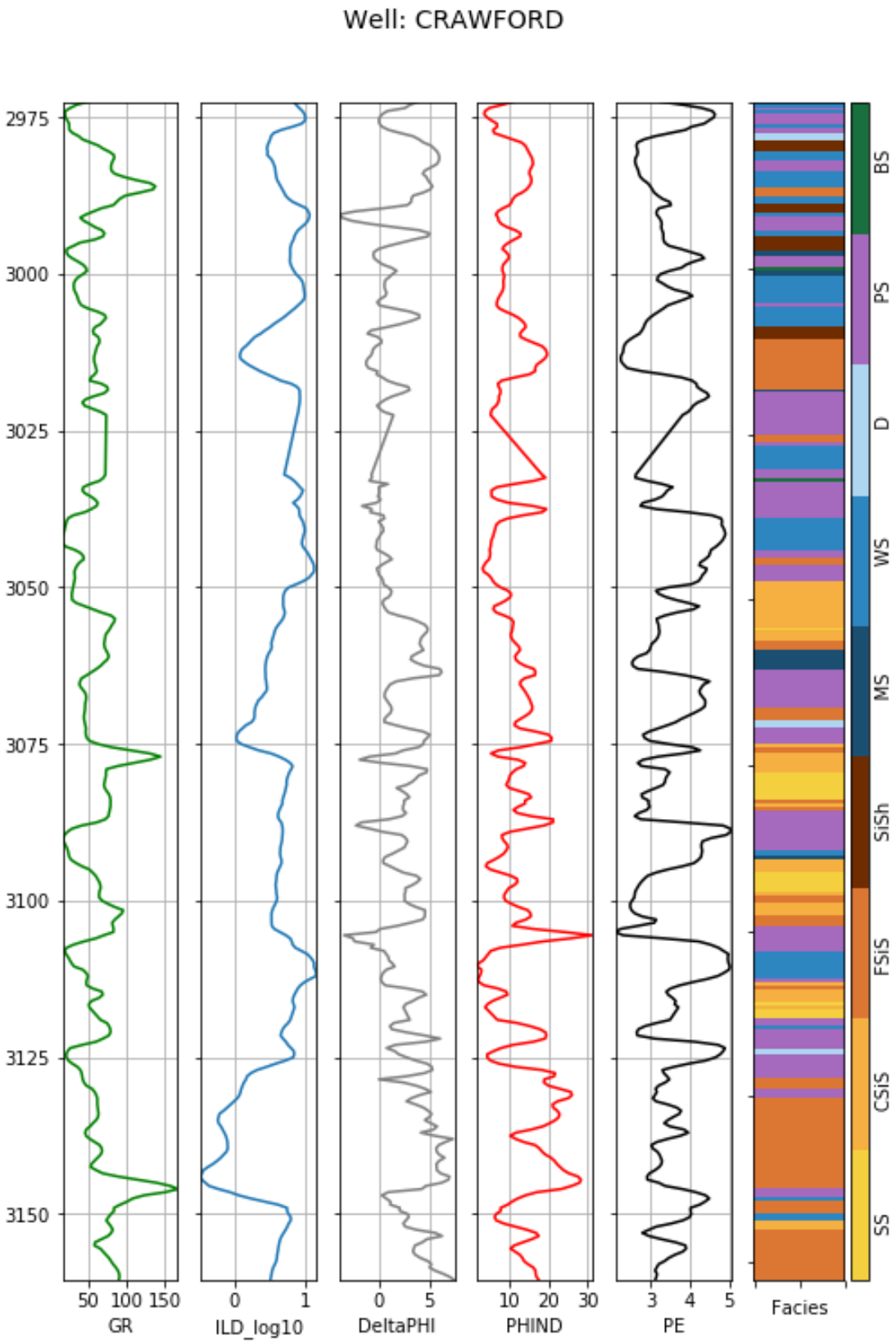
```
[STUART, CRAWFORD]
Categories (2, object): [STUART, CRAWFORD]
```

We can use the well log plot to view the classification results along with the well logs.

In [43]:

```
make_facies_log_plot(
    well_data[well_data['Well Name'] == 'STUART'],
    facies_colors=facies_colors)

make_facies_log_plot(
    well_data[well_data['Well Name'] == 'CRAWFORD'],
    facies_colors=facies_colors)
```


Well: STUART
2850
2900
2950
3000
100
200
GR
0.5
1.0
1.5
ILD_log10
0
10
DeltaPHI
10
20
PHIND
4
6
PE
Facies
BS
PS
D
WS
MS
SiSh
FSiS
CSiS
SS


Well: CRAWFORD
2975
3000
3025
3050
3075
3100
3125
3150
50 100 150
GR
0 1
ILD_log10
0 5
DeltaPHI
10 20 30
PHIND
3 4 5
PE
Facies
BS
PS
D
WS
MS
SiSh
FSiS
CSiS
SS

Finally we can write out a csv file with the well data along with the facies classification results.

In [44]:

```
well_data.to_csv('well_data_with_facies.csv')
```

References

Amato del Monte, A., 2015. Seismic Petrophysics: Part 1, *The Leading Edge*, 34 (4). doi:10.1190/tle34040440.1 (http://dx.doi.org/10.1190/tle34040440.1)

Bohling, G. C., and M. K. Dubois, 2003. An Integrated Application of Neural Network and Markov Chain Techniques to Prediction of Lithofacies from Well Logs, *KGS Open-File Report* 2003-50, 6 pp. pdf (http://www.kgs.ku.edu/PRS/publication/2003/ofr2003-50.pdf)

Dubois, M. K., G. C. Bohling, and S. Chakrabarti, 2007, Comparison of four approaches to a rock facies classification problem, *Computers & Geosciences*, 33 (5), 599-617 pp. doi:10.1016/j.cageo.2006.08.011 (http://dx.doi.org/10.1016/j.cageo.2006.08.011)