5/26/2018

Manual for Geological Log Data Machine Learning with Python

Syed Muhammad Amir

TEXAS TECH UNIVERSITY

Acknowledgment

First, I would like to express my deepest gratitude to my advisor Dr. Ekarit Panacharoensawad, Assistant Professor, Department of Petroleum Engineering, Texas Tech University for his constant support throughout this work. During the period I have worked under him, I believe I have learned more than I have ever had working with any other mentors in my life.

Dr. Ekarit's constant belief in the fact that anyone can achieve anything unless that individual is willing to work hard as he/she can, was the factor that drove me to the path of programming and then Machine Learning which I never thought to take in my life, coming from a zero-programming background. I cannot thank him enough for his continuous belief in me and moreover his unconditioned sincerity, responsibility and commitment to the cause of pushing me towards achieving my professional goals throughout my journey at Texas Tech University.

I would also like to thank Dr. Henderson for sharing the data to us for this work along with his effort in extracting more information out of the data than was provided and sharing his geological wisdom with us that helped in achieving the goals of the project.

My family has played the most important role in becoming a man I am today. Without their constant support and love in all phases of my personal and professional struggles and having their faith in the fact that I can become a better version of my own self with each day I wouldn't be where I am today.

Finally, and most importantly, all this was not possible without God's love for all His creation. I have faced a tough time in my personal and professional life and when I look back at how everything that He had planned worked out to be where I am today, I realize how shortsighted we as a human can be when we are devastated when our own wishes are not met. Indeed, what He has planned for us is the best for us all aspects of our life.

"*Persevere and endure and remain patient and fear God that you may be successful.*"

1.1: Abstract

Formation Evaluation is a part of geological science dedicated to studying log data obtained from different electric logs which are sent in the subsurface to gets important information about the types of formations and whether it has sufficient oil & gas reserves to cover the economics of the project or not.

Over the year, a dedicated team of geologist works on analyzing the data from the logs and make predictions about the type of formation and whether the results from the logs shows a prospective oil/gas zone at several intervals of depth inside the formation. The process has always been tricky and requires attention to each detail.

Despite the success of the process over the year, we still have higher chances of misinterpreting the logs that ultimately results in loss of millions of dollars of a company in a project that turned out to be a dry hole or missed prospective zone etc., based on the misjudged log interpretation even by experienced personnel.

With the rise of Artificial Intelligence (A.I) and its one of the most popular tools i.e., Machine Learning (ML) this issue can be addressed very well. ML involves using a set of data that involves training the system by use of ML algorithms that finds patterns within the data that may have not been captured by human earlier and automatically learn and improve from experience with more data without requirement of being explicitly programmed. Conclusively, when a new set of data is fed to the machine it would be able to make predictions based on the algorithm's learning about the data during its training.

We will use ML algorithms in this project to train the system on the geological log data and feed it some targets (i.e., types of formations based in this case), and use this technique (called *supervised learning*) to make predictions of the type of formation on the new data that the algorithm never saw.

Table of Contents

2.1: Introduction

2.1.1 : Some basics about Machine Learning

Machine Learning involves the study of pattern recognition and computational learning theory in artificial intelligence. It requires to study the data and develop algorithms in such a way that a system is able to learn and make predictions on data. Its earliest applications have been dated all the way back to 1959 and since then it has evolved in many forms and today it is the foundation of many of our predictive tech products such as ranking web searches, smartphone's recognition, recommending videos on YouTube, and so on.

2.1.2 : Machine Learning in Petroleum Engineering applications

As machine learning finds its place in many engineering and technological applications of the world, it seems to be a practical approach to find a way to incorporate this artificial intelligence system in the oil & gas industry as well.

One of the many areas of the petroleum industry where it can be immediately introduced is formation evaluation from the geological log data. Several logs are run down the hole (caliper, neutron, bulk density, photoelectric logs, etc.) and a qualitative & quantitative analysis is done by a team of geologist from the information these logs bring from down the hole to decide the type of formations and the prospective formations which contains oil & gas.

During the interpretation phase of the logs, extreme concentration and attention to details are required so that an important piece of information is not misinterpreted or missed. If this is not done, a company can be hit by a loss of millions of dollars in the investment of drilling through the hole and then running many expensive logs just to find out that a geologist didn't pay attention and misinterpreted a formation or missed a prospective zone.

With machine learning algorithms, training datasets with information like types of formation & type of produced fluid etc., we can eliminate the human error factor in no time and automate the process of interpretation of logs and save millions of dollars of a company on buggy interpretation.

This kind of work has never been done before in the petroleum industry and is going to be possibly one of the many of the first implementation of machine learning on the geological log data in the oil & gas energy sector. The goal of this project is to make the formation evaluation process easier, accurate and more automated.

3.1: Training the system

3.1.1 : About the data

A geological log data obtained through the courtesy of Dr. Steven Henderson, Associate Professor of Practice, Department of Petroleum Engineering, Texas Tech University was used for training the system. It contains data from commonly used mechanically/electrically operated logs such as Caliper, Neutron, Density, Resistivity, Sonic and others that brings information at every foot of the formation where it is passing through. The data is of a depth of approximately 2000-feet interval which was used to predict the type of formation.

The log data belongs to a well named "*RUFENACHT",* in *"Ness"* county of Hays, Kansas (Kanas Geological Survey, 2018). The Ness county is mostly all carbonaceous formations with dominance of formations such as limestone, shaly limestone, dolomite and shales with streaks of sandstone and shaly sandstone here and there (as you will see later when we visualize the data).

Though the data that we obtained from Dr. Henderson was from a public domain of Kanas Geological Survey where there are log data of similar types available from the wells drilled in the Kansas state, interpretation of the type of formation from the log data isn't available with any of the data on the domain.

Thanks to Dr. Henderson who spent a great deal of his time interpreting the type of formation from the log data file he shared with us of each cell individually. This made our job a lot easier as we had the targets (*types of formation)* available to be fed to ML algorithm because interpreting the type of formation from the log data is a time-consuming task itself. These targets were then fed to the ML algorithms for training (called *supervised machine learning*).

Once the system is finally trained on the available numerical data with their targets by implementing ML algorithms, we would be able to predict with a range of accuracy that this is the type of formation in the case when combined given log values lies within these ranges.

3.1.2 : Implementing ML on the data

3.1.2.1 : Overview the data

Before implementing any ML algorithms to train and make predictions on the data, it is very important to have a quick look of how the raw data looks like. Python (language used in this project to implement ML

on the data) has many libraries that allow us to load the data, visualize it and train ML algorithms. We have used the famous PANDAS along with NUMPY libraries to read the Excel file that contains the data, clean the data, followed by MATPLOTLIB libraries to visualize the data and several of Scikit-Learn's ML algorithms that will be discussed later and shown along with the codes. We begin with importing the most common libraries that will be used throughout as can be seen:

```
In [1]: # Common imports
        import numpy as np
        import pandas as pd
        # to make this notebook's output stable across runs
        np.random.seed(42)

        # To plot pretty figures
        %matplotlib inline
        import matplotlib
        import matplotlib.pyplot as plt
        plt.rcParams['axes.labelsize'] = 14
        plt.rcParams['xtick.labelsize'] = 12
        plt.rcParams['ytick.labelsize'] = 12
```

Figure 1: Common imports

To have Python have a quick overlook of the tables & their values in the excel file we would import the Pandas library and make it read the excel file. Once it has read the file, without altering the original excel file we can view the data before and after cleaning:

```
In [2]: LogDat = pd.read_excel(r"C:/Users/Amir's/Desktop/Amir_Adjusted.xlsx")
```

Figure 2: Pandas read Excel

```
[3]: LogDat
```

Bulk density	Density Correction	Resistivity (Deep)	Resistivity (Medium)	...	Micro-inverse resistivity (micro log)	Micro-normal resistivity (micro log)	Delta-t (interval transit time, or slowness)	Sonic porosity	Type of Formation	Unnamed: 18	NPOR	Neutron porosity, calculated assuming a limestone matrix	Unnamed: 21	Unnamed: 22
2.2138	-0.0356	0.9126	1.0719	...	5.1127	7.7722	78.7252	22.0122	shaly limestone	NaN	CALI	Caliper	NaN	NaN
2.2234	-0.0395	0.8803	1.0008	...	5.0602	7.4297	78.2474	21.6743	shaly limestone	NaN	DPOR	Density porosity, calculated assuming a limest...	NaN	NaN
2.2424	-0.0362	0.8754	0.9679	...	4.9294	7.0917	77.6106	21.2239	shaly limestone	NaN	GR	Gamma	ray	NaN
2.2766	-0.0289	0.9005	0.9813	...	5.2303	7.0816	76.7257	20.5981	shaly limestone	NaN	PE	Photoelectric	factor	NaN
									shaly					

Figure 3: Overview log data

As we can see from the codes and their results from above, before cleaning the data there are some unwanted extra columns filled with Not A Number(NANs) that would not be needed at all for training the data and in fact would slow down the training and give erroneous results. Also, we would delete these columns right away before even visualizing the doing the following:

```
In [4]: LogDat_adj = LogDat.drop(LogDat.columns[[18,19,20,21,22]], axis=1)  # df.columns is zero-based pd.Index
In [5]: LogDat_adj
```

Gamma ray	Photoelectric	Bulk density	Density Correction	Resistivity (Deep)	Resistivity (Medium)	Resistivity (Shallow)	Ratio (shallow/deep resistivity)	Spontaneous Potential	Micro-inverse resistivity (micro log)	Micro-normal resistivity (micro log)	Delta-t (interval transit time, or slowness)	Sonic porosity	Type of Formation
26.4565	4.0462	2.2138	-0.0356	0.9126	1.0719	5.2530	-68.4118	-20.3987	5.1127	7.7722	78.7252	22.0122	shaly limestone
28.7921	4.1226	2.2234	-0.0395	0.8803	1.0008	4.6464	-65.0243	-19.9382	5.0602	7.4297	78.2474	21.6743	shaly limestone
27.4413	4.2350	2.2424	-0.0362	0.8754	0.9679	4.3056	-62.2656	-19.4078	4.9294	7.0917	77.6106	21.2239	shaly limestone
25.6896	4.3685	2.2766	-0.0289	0.9005	0.9813	4.1801	-60.0036	-18.7673	5.2303	7.0816	76.7257	20.5981	shaly limestone
27.0588	4.5133	2.3217	-0.0250	0.9582	1.0502	4.1355	-57.1585	-17.8640	5.2853	7.2144	75.4503	19.6961	shaly limestone
30.6761	4.6562	2.3681	-0.0284	1.0526	1.1877	4.8743	-59.9082	-16.5961	5.2074	7.8358	73.6905	18.4515	shaly limestone

Figure 4: Remove extra columns

With that, we have removed extra columns and are now ready to learn more about the data and then visualize it:

```
In [6]: LogDat_adj.info()
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1989 entries, 0 to 1988
Data columns (total 18 columns):
DEPTH                                              1989 non-null float64
Neutron Porosity                                   1989 non-null float64
Caliper                                            1989 non-null float64
Density Porosity                                   1989 non-null float64
Gamma ray                                          1989 non-null float64
Photoelectric                                      1989 non-null float64
Bulk density                                       1989 non-null float64
Density Correction                                 1989 non-null float64
Resistivity (Deep)                                 1989 non-null float64
Resistivity (Medium)                               1989 non-null float64
Resistivity (Shallow)                              1989 non-null float64
Ratio (shallow/deep resistivity)                   1989 non-null float64
Spontaneous Potential                              1989 non-null float64
Micro-inverse resistivity (micro log)              1989 non-null float64
Micro-normal resistivity (micro log)               1989 non-null float64
Delta-t (interval transit time, or slowness)       1989 non-null float64
Sonic porosity                                     1989 non-null float64
Type of Formation                                  1936 non-null object
dtypes: float64(17), object(1)
memory usage: 272.0+ KB
```

Figure 5: Basic information of the data

```
In [286]: LogDat_adj.hist(bins=50, figsize=(20,15))
```

Figure 6:Histogram of numerical values

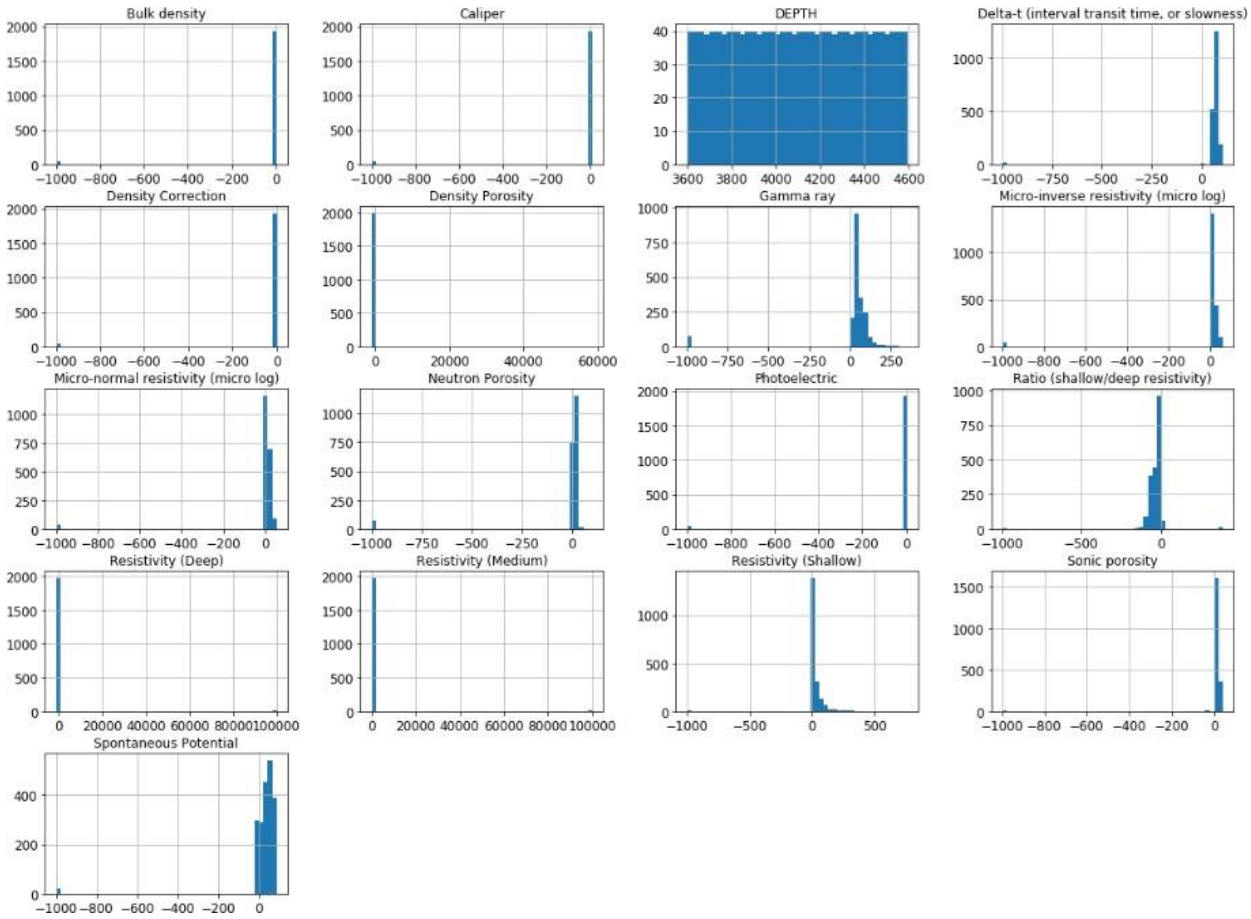

Figure 7: Histogram of numerical values

The .hist () method comes from the pandas library and indicates the usual frequency of the values of each feature in the table (all numerical). As we can see, most of the data lies within a certain range except for some unusually off points that lies around -1000. Upon going back to the tables already seen earlier, we see there are points indicating -999.25 which maybe because when the log tool is starting initially it takes some erroneous reading before starting to normalize again. These points will be dealt with in data cleaning part.

Also, we can see that there are about 2000 cells in total and *"Types of Formation*" attribute has some values missing towards the end of it (1936 cells instead of 1989), which also needs to be taken care of.

Some other information can also be seen about the data such as the total number of each type of formation that exist in the subsurface and sum statistics related to the numerical attributes of the data:

```
In [7]: LogDat_adj["Type of Formation"].value_counts()

Out[7]: limestone          956
        shaly limestone    456
        shale              258
        dolomite           163
        sandstone           59
        sandy limestone     38
        shaly sandstone      6
        Name: Type of Formation, dtype: int64
```

Figure 8: Counting total values of each type of formations

```
In [8]: LogDat_adj.describe()
```

Out[8]:

	DEPTH	Neutron Porosity	Caliper	Density Porosity	Gamma ray	Photoelectric	Bulk density	Density Correction	Resistivity (Deep)	Resistivity (Medium)	Resistivity (Shallow)
count	1989.000000	1989.000000	1989.000000	1989.000000	1989.000000	1989.000000	1989.000000	1989.000000	1989.000000	1989.000000	1989.000000
mean	4097.000000	-20.183288	-17.790037	15.149971	16.863204	-21.745637	-23.678348	-26.110950	611.082785	668.888914	26.364424
std	287.159581	185.811695	159.254436	1323.847169	210.172649	158.613934	159.884135	159.485422	7746.001211	8058.989422	80.716629
min	3600.000000	-999.250000	-999.250000	-999.250000	-999.250000	-999.250000	-999.250000	-999.250000	-999.250000	0.933400	-999.250000
25%	3848.500000	9.126500	7.824600	5.306900	33.252500	3.435300	2.411800	-0.027600	3.207600	3.463900	6.482900
50%	4097.000000	14.206000	7.963200	10.149200	45.027200	4.119000	2.525400	-0.004600	6.791700	7.057600	12.195300
75%	4345.500000	19.510700	8.116000	16.641000	72.440300	4.509200	2.611900	0.032700	11.986000	13.988200	26.418100
max	4594.000000	100.000000	10.674400	58594.152300	351.118300	5.908800	3.760500	0.250300	100000.000000	100000.000000	761.316400

Figure 9: Numerical description of the data

The .describe() method gives numerical description of the data for an overlook on means, standard deviation (std), min, max and mean values and the Percentile ranks (25%, 50% 75%).

The data overviewing can be important in finding out patterns from human eyes as well. As we may see that certain log values lies within certain ranges to give a predicted target to be a certain type of formations, but they may not be necessarily true, OR can be much more than that so we can never be sure until we further investigate the data and train the data by implementing ML algorithms on it. Since, we now gone through a quick overview of the data and can move to the data cleaning part.

3.1.2.2 : Data Cleaning

As we saw in the overview that there are some points in the data that need to be taken care of before training the data as they would lead to bad training of the data and poor predictions. To deal with the off-points as seen in the plots, we would find the indices of the cells with those values by looking at the columns where it has started to occur the earliest. This is shown below:

```
In [10]: LogDat_prob_GR = LogDat_adj[(LogDat_adj['Gamma ray'] <= -700)]
         LogDat_prob_GR.head()
```

Out[10]:

	DEPTH	Neutron Porosity	Caliper	Density Porosity	Gamma ray	Photoelectric	Bulk density	Density Correction	Resistivity (Deep)
1910	4555.0	26.5237	7.6225	20.0142	-999.25	2.6090	2.3678	0.0404	2.5194
1911	4555.5	26.6968	7.6137	19.7698	-999.25	2.5529	2.3719	0.0520	2.4917
1912	4556.0	27.3526	7.5680	19.5087	-999.25	2.5357	2.3764	0.0522	2.4647
1913	4556.5	28.3447	7.5731	19.3305	-999.25	2.5532	2.3794	0.0462	2.4469
1914	4557.0	33.4406	7.6147	19.8108	-999.25	2.5530	2.3712	0.0368	2.4500

Figure 10: Finding index of off-points

Before removing the indices where the -999.25 values started to occur, we also need to find the indices of the null values of "Type of Formations" column as discussed earlier.

```
In [9]: inds = pd.isnull(LogDat_adj['Type of Formation']).nonzero()[0]
        inds
```

```
Out[9]: array([1936, 1937, 1938, 1939, 1940, 1941, 1942, 1943, 1944, 1945, 1946,
               1947, 1948, 1949, 1950, 1951, 1952, 1953, 1954, 1955, 1956, 1957,
               1958, 1959, 1960, 1961, 1962, 1963, 1964, 1965, 1966, 1967, 1968,
               1969, 1970, 1971, 1972, 1973, 1974, 1975, 1976, 1977, 1978, 1979,
               1980, 1981, 1982, 1983, 1984, 1985, 1986, 1987, 1988], dtype=int32)
```

Figure 11: Finding indices of missing attributes

We can now safely remove these rows:

```
In [12]: LogDat_drop = LogDat_adj.drop(LogDat_adj.index[1905:1989])
```

Figure 12: Dropping missing attributes and off-points

Besides the way we took care of the null values of the attribute "Types of Formation" there is an alternative way of dealing with such values which involves taking the median of the data in that missing attribute by calling Scikit-Learn's "*Imputer*", but imputer would work best in case we are dealing with numerical attributes only. In the case of our data, since we are dealing with missing values in the target class "Types of formation" which involves text attributes, and even though it would be later converted to numerical values as discussed later, it would still be a bad idea as the values would be in Boolean form.

3.1.2.3 : Creating test and training sets

3.1.2.3.1 What are training and test sets?

Before visualizing the data further or doing anything more with the data we need to split the data into training & test sets and never looking at the test set again and training the model only on the training set. This is a common and important practice adopted in ML in order that we can see how well we trained our models on a part of data and then evaluate the performance on the unseen part of the data (test set). If test set is used while training, this would sort of cheating because test set is only used for evaluation purpose at the end of training our models and then generalizing on a totally new data would proofthat we cheated using the test set during the training of model.

Another good practice is the *cross-validation* that you would find in the algorithms over and over here which is basically training set split into complementary subsets, and each model is trained against a different combination of the subsets and is validated with the remaining parts of the data.

A common practice is to split the data in such a way that the training set is 80% of the total dataset, while the rest is the test set.

3.1.2.3.2 : Algorithm to split the training and test sets:

Since, the data we are dealing is comparatively small, we would opt to use yet another technique to splitting the data into training and test sets. If we would try the conventional ML's Scikit-Learn library's "*train_test_split*", we find that the test set has missing attributes categories of the "Types of formation" feature. This means, the test set is not the true representative of all the categories of the data. To address

this, we use Scikit-Learn's another technique to split the data categorically called "*StratifiedShuffleSplit*". Here's the algorithm we implemented:

```
In [287]: from sklearn.model_selection import StratifiedShuffleSplit

          split = StratifiedShuffleSplit(test_size=0.27, random_state=42)
          for train_index, test_index in split.split(LogDat_drop, LogDat_drop["Type of Formation"]):
              strat_train_set = LogDat_drop.loc[train_index]
              strat_test_set = LogDat_drop.loc[test_index]
```

Figure 13: Creating training and test sets using Shuffled-split technique

The algorithm implemented shows that the size of the training set was 27%, while the rest is the training set, and that the data was split considering all the categories in our target class "Types of Formation" are addressed.

Initially, we decided to go with the conventional 0.8:0.2 ratio of training to test sets but we found that there were none of the "shaly sandstone" class in the "Types of Formations" target class that occurred in the test set despite *StratifiedShuffleSplit.* For this, we had to increase the size of the test set to 27% and we had then only 1 instance out of the total 6 instances of "shaly sandstone" class that occurred in the test set. This is still not an ideal condition as predicting only on 1 instance of the class would be fine, but we can never know how accurate the training was when more instances are introduced.

To increase the number of instances of "shaly sandstone" in the test set, we found a better way to split the data courtesy of *Eduard Ilyasov* reply to a similar problems as ours *Stackoverflow* who suggested that assuming that we generate concatenated dataset from our train and test sets, get dummies (also known as Boolean indicators which we will see in the next part why it was helpful for text attributes classification problem like ours') from the concatenated dataset and split it back to training and test sets.

```
In [17]: train_objs_num = len(strat_train_set)
         train_objs_num
```

Figure 14: Finding length of stratified training set

```
In [19]: dataset = pd.concat(objs=[strat_train_set, strat_test_set], axis=0)
         dataset.head()
```

Out[19]:

	DEPTH	Neutron Porosity	Caliper	Density Porosity	Gamma ray	Photoelectric	Bulk density	Density Correction	Resistivity (Deep)
364	3782.0	14.9090	8.0845	10.8138	37.9302	4.3361	2.5251	-0.0403	2.3521
1119	4159.5	5.4242	7.9526	2.0286	87.9380	4.9899	2.6753	-0.0504	12.3318
974	4087.0	26.8415	7.6343	30.5027	30.2045	4.2036	2.1884	-0.0213	23.6848
481	3840.5	29.6743	10.0896	17.5728	155.6782	3.4528	2.4095	0.1149	2.1401
828	4014.0	3.9354	8.1300	3.0607	38.1546	4.9711	2.6577	-0.0481	28.2380

Figure 15: Combining stratified splits of training and test sets

```
In [20]: dataset_preprocessed = pd.get_dummies(dataset)
         dataset_preprocessed.head()
```

Out[20]:

ılicro- ɔrmal ɨtivity ɲicro log)	Delta-t (interval transit time, or slowness)	Sonic porosity	Type of Formation_dolomite	Type of Formation_limestone	Type of Formation_sandstone	Type of Formation_sandy limestone
.8936	68.4875	14.7719	0	1	0	0
.7671	53.3764	4.0851	0	1	0	0
.4710	70.9938	16.5444	0	0	0	1
.0344	91.0397	30.7212	0	0	0	0
.2133	54.2995	4.7380	0	1	0	0

Figure 16: Converting to Boolean by get_dummies

```
In [21]: train_preprocessed = dataset_preprocessed[:train_objs_num]
         train_preprocessed.head()
```

Out[21]:

Micro-normal stivity (micro log)	Delta-t (interval transit time, or slowness)	Sonic porosity	Type of Formation_dolomite	Type of Formation_limestone	Type of Formation_sandstone	Type of Formation_sandy limestone
5.8936	68.4875	14.7719	0	1	0	0
7.7671	53.3764	4.0851	0	1	0	0
4.4710	70.9938	16.5444	0	0	0	1
2.0344	91.0397	30.7212	0	0	0	0
3.2133	54.2995	4.7380	0	1	0	0

Figure 17: Splitting the part of the data into training set after converting into dummies

```
In [22]: test_preprocessed = dataset_preprocessed[train_objs_num:]
         test_preprocessed.head()
```

Out[22]:

Micro-normal stivity micro log)	Delta-t (interval transit time, or slowness)	Sonic porosity	Type of Formation_dolomite	Type of Formation_limestone	Type of Formation_sandstone	Type of Formation_sandy limestone
.5789	70.6689	16.3146	0	1	0	0
.2743	73.7622	18.5023	0	1	0	0
.6692	68.4833	14.7690	0	0	0	0
.0720	61.6092	9.9075	0	1	0	0
.8596	92.2882	31.6041	0	0	0	0

Figure 18: Splitting the part of the data into test set after converting into dummies

Ultimately, the splitting of the data this way also converted the categorial attributes to Boolean form (by getting the dummy variables) and solved another problem at the same time (discussed later).

Now we can separate the training and test sets from the tables using some Pandas functions and verify if all classes actually appeared in the target class.

```
LogDat_train_num = train_preprocessed.drop(train_preprocessed.loc[:,'Type of Formation_dolomite':
                                                                  'Type of Formation_shaly sandstone'].head(0).columns, axis=1)
LogDat_train_num.head()
```

	DEPTH	Neutron Porosity	Caliper	Density Porosity	Gamma ray	Photoelectric	Bulk density	Density Correction	Resistivity (Deep)	Resistivity (Medium)	Resistivity (Shallow)	Ratio (shallow/deep resistivity)	Spontaneous Potential	Mi inv resist (m
364	3782.0	14.9090	8.0845	10.8138	37.9302	4.3361	2.5251	-0.0403	2.3521	2.4766	7.0462	-42.8840	6.5761	18.2
1119	4159.5	5.4242	7.9526	2.0286	87.9380	4.9899	2.6753	-0.0504	12.3318	15.4253	34.3674	-40.0607	19.4222	36.6
974	4087.0	26.8415	7.6343	30.5027	30.2045	4.2036	2.1884	-0.0213	23.6848	24.9051	67.3733	-40.8617	-5.8343	8.5
481	3840.5	29.6743	10.0896	17.5728	155.6782	3.4528	2.4095	0.1149	2.1401	2.1240	2.8021	-10.5339	63.1526	2.5
828	4014.0	3.9354	8.1300	3.0607	38.1546	4.9711	2.6577	-0.0481	28.2380	39.9660	162.5017	-68.4021	40.2980	46.6

Figure 19: Creating numerical training set

```
LogDat_train_labels = train_preprocessed.drop(train_preprocessed.loc[:,'DEPTH':
                                                                     'Sonic porosity'].head(0).columns, axis=1)
LogDat_train_labels.head()
```

	Type of Formation_dolomite	Type of Formation_limestone	Type of Formation_sandstone	Type of Formation_sandy limestone	Type of Formation_shale	Type of Formation_shaly limestone	Type of Formation_shaly sandstone
364	0	1	0	0	0	0	0
1119	0	1	0	0	0	0	0
974	0	0	0	1	0	0	0
481	0	0	0	0	1	0	0
828	0	1	0	0	0	0	0

Figure 20: Creating targets for training set

```
LogDat_test_num = test_preprocessed.drop(test_preprocessed.loc[:,'Type of Formation_dolomite':
                                                               'Type of Formation_shaly sandstone'].head(0).columns, axis=1)
LogDat_test_num.head()
```

	DEPTH	Neutron Porosity	Caliper	Density Porosity	Gamma ray	Photoelectric	Bulk density	Density Correction	Resistivity (Deep)	Resistivity (Medium)	Resistivity (Shallow)	Ratio (shallow/deep resistivity)	Spontaneous Potential	Mi inv resist (m
1501	4350.5	14.3377	7.9358	8.1210	53.0320	4.1425	2.5711	-0.0126	8.2467	8.7727	13.0996	-18.0879	83.4069	14.2
377	3788.5	21.5996	7.7935	20.2126	31.8369	3.9984	2.3644	-0.0032	1.3436	1.4921	5.5935	-55.7457	-7.8881	6.3
1025	4112.5	9.2560	8.0716	6.8427	63.5457	3.9832	2.5930	0.1195	8.1046	7.6471	26.1792	-45.8304	59.9619	19.7
819	4009.5	9.8961	8.1299	6.8641	45.8450	4.8488	2.5926	-0.0653	18.2810	18.2213	26.1286	-13.9605	52.6570	23.2
1364	4282.0	25.5932	9.6504	32.0067	179.7232	2.5904	2.1627	0.1948	4.4241	3.5417	4.1200	2.7839	71.7486	2.5

Figure 21: Creating numerical test set

```
In [29]: LogDat_test_labels = test_preprocessed.drop(test_preprocessed.loc[:,'DEPTH':
                                                                'Sonic porosity'].head(0).columns, axis=1)
         LogDat_test_labels.head()
```

Out[29]:

	Type of Formation_dolomite	Type of Formation_limestone	Type of Formation_sandstone	Type of Formation_sandy limestone	Type of Formation_shale	Type of Formation_shaly limestone	Type of Formation_shaly sandstone
1501	0	1	0	0	0	0	0
377	0	1	0	0	0	0	0
1025	0	0	0	0	1	0	0
819	0	1	0	0	0	0	0
1364	0	0	0	0	1	0	0

Figure 22: Creating targets for test set

```
LogDat_test_labels.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 515 entries, 1501 to 368
Data columns (total 7 columns):
Type of Formation_dolomite           515 non-null uint8
Type of Formation_limestone          515 non-null uint8
Type of Formation_sandstone          515 non-null uint8
Type of Formation_sandy limestone    515 non-null uint8
Type of Formation_shale              515 non-null uint8
Type of Formation_shaly limestone    515 non-null uint8
Type of Formation_shaly sandstone    515 non-null uint8
dtypes: uint8(7)
memory usage: 7.5 KB
```

Figure 23: Getting information about the test set

```
LogDat_test_labels.sum()
```

```
Type of Formation_dolomite            36
Type of Formation_limestone          258
Type of Formation_sandstone           16
Type of Formation_sandy limestone     10
Type of Formation_shale               70
Type of Formation_shaly limestone    123
Type of Formation_shaly sandstone      2
dtype: int64
```

Figure 24: Checking total values of each type of formation in the test set

```
LogDat_train_labels.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 1390 entries, 364 to 1301
Data columns (total 7 columns):
Type of Formation_dolomite           1390 non-null uint8
Type of Formation_limestone          1390 non-null uint8
Type of Formation_sandstone          1390 non-null uint8
Type of Formation_sandy limestone    1390 non-null uint8
Type of Formation_shale              1390 non-null uint8
Type of Formation_shaly limestone    1390 non-null uint8
Type of Formation_shaly sandstone    1390 non-null uint8
dtypes: uint8(7)
memory usage: 20.4 KB
```

Figure 25: Getting information of the training set

```
LogDat_train_labels.sum()
```

```
Type of Formation_dolomite           96
Type of Formation_limestone         698
Type of Formation_sandstone          43
Type of Formation_sandy limestone    28
Type of Formation_shale             188
Type of Formation_shaly limestone   333
Type of Formation_shaly sandstone     4
dtype: int64
```

Figure 26: Checking total number of each attribute in types of formation to train in training set

Before assigning the values to X's and y's trains and test. It is important to normalize the data before being used in training. This is discussed in the next section.

3.1.2.3.3 : *Feature Scaling*

Generally, with a few exceptions ML algorithms don't do very well when the input numerical values have very different scales as we can see that in our case too that depth has a very different scale than deep/shallow ration or Spontaneous Potential (SP log) with Caliper log etc. If the ML algorithms are trained on this, they would definitely result in poor training and thus poor predictions.

To address this, there are two types of scalers most commonly used:

1- Min-max scaler
2- Standardization

In the min-max scaler, values are shifted and rescaled which results in the values ending up between 0-1. This is accomplished by subtracting minimum value and dividing by the max minus the min.

On the other hand, standardization is achieved by subtracting the mean value (resulting in the standardized values always having a zero mean), and then it divides it by the variance so that the resulting distribution has a unit variance.

$$x' = \frac{x - \mu}{\sigma}$$

Figure 27: Variance formula for Standard Scaling, Geron (2017)

Since, standardization is much more sensitive to outliers, we have used the Scikit-Learn's *StandardScaler* for this purpose. Its use has been shown on our data below:

```
from sklearn.preprocessing import StandardScaler

std_scaler = StandardScaler()
LogDat_train_copy = LogDat_train_num.copy()
LogDat_train_scaled = std_scaler.fit_transform(LogDat_train_copy)
LogDat_train_scaled
```

```
array([[ -1.05005254e+00,   1.66138577e-03,   9.97496737e-02, ...,
          2.48495548e-01,  -1.97490010e-02,  -1.97525730e-02],
       [  3.25716993e-01,  -1.33346182e+00,  -2.65951524e-01, ...,
          1.55079941e+00,  -1.46864611e+00,  -1.46865324e+00],
       [  6.14963541e-02,   1.68133388e+00,  -1.14845866e+00, ...,
          9.24625785e-02,   2.20562481e-01,   2.20560377e-01],
       ...,
       [  1.68508663e+00,   5.76360644e-01,  -6.94867182e-01, ...,
         -4.15132656e-01,   1.29627074e-01,   1.29627855e-01],
       [  1.59033164e+00,   3.27573452e-01,  -1.32978435e+00, ...,
         -2.60448770e-01,  -5.68468048e-01,  -5.68466011e-01],
       [  6.57359450e-01,  -7.82045015e-01,  -2.07450423e-01, ...,
          4.80340404e-01,  -5.75275738e-01,  -5.75285611e-01]])
```

Figure 28: Standard scaling the data

```
X_train = LogDat_train_scaled
```

```
X_train.shape
```

```
(1390, 17)
```

```
X_train
```

```
array([[ -1.05005254e+00,   1.66138577e-03,   9.97496737e-02, ...,
          2.48495548e-01,  -1.97490010e-02,  -1.97525730e-02],
       [  3.25716993e-01,  -1.33346182e+00,  -2.65951524e-01, ...,
          1.55079941e+00,  -1.46864611e+00,  -1.46865324e+00],
       [  6.14963541e-02,   1.68133388e+00,  -1.14845866e+00, ...,
          9.24625785e-02,   2.20562481e-01,   2.20560377e-01],
       ...,
       [  1.68508663e+00,   5.76360644e-01,  -6.94867182e-01, ...,
         -4.15132656e-01,   1.29627074e-01,   1.29627855e-01],
       [  1.59033164e+00,   3.27573452e-01,  -1.32978435e+00, ...,
         -2.60448770e-01,  -5.68468048e-01,  -5.68466011e-01],
       [  6.57359450e-01,  -7.82045015e-01,  -2.07450423e-01, ...,
          4.80340404e-01,  -5.75275738e-01,  -5.75285611e-01]])
```

Figure 29: Assigning the standard scaled value to training

```
X_test_new = LogDat_test_num.copy()
X_test_new.head()
```

	DEPTH	Neutron Porosity	Caliper	Density Porosity	Gamma ray	Photoelectric	Bulk density	Density Correction	Resistivity (Deep)
1501	4350.5	14.3377	7.9358	8.1210	53.0320	4.1425	2.5711	-0.0126	8.2467
377	3788.5	21.5996	7.7935	20.2126	31.8369	3.9984	2.3644	-0.0032	1.3436
1025	4112.5	9.2560	8.0716	6.8427	63.5457	3.9832	2.5930	0.1195	8.1046
819	4009.5	9.8961	8.1299	6.8641	45.8450	4.8488	2.5926	-0.0653	18.2810
1364	4282.0	25.5932	9.6504	32.0067	179.7232	2.5904	2.1627	0.1948	4.4241

Figure 30: Copying the test set so that no harm is done to original test set

```
X_test_prepared = std_scaler.transform(X_test_new)
X_test_prepared
```

```
array([[ 1.02180171, -0.07875738, -0.3125306 , ..., -0.22474743,
         0.1894101 ,  0.18940443],
       [-1.02636379,  0.94346041, -0.70706646, ..., -0.47751952,
         0.48600489,  0.48600959],
       [ 0.15442913, -0.79408038,  0.0639836 , ...,  0.22388301,
        -0.02015171, -0.02014575],
       ...,
       [ 1.63770913, -0.5776831 , -0.74920944, ...,  0.23111102,
        -1.01128436, -1.01127906],
       [ 1.2787335 ,  1.87064807, -0.8188008 , ..., -1.27187716,
         3.28191387,  3.28191009],
       [-1.0427637 , -0.06266799,  0.11777134, ..., -0.44686353,
         0.18954434,  0.18954001]])
```

Figure 31: Transforming the test set by standard scaling and assigning it to be a part of the test set

Note that we have used fit_transform () in the training set, while just fit() in the test set. The logic behind is that when we call fit_transform (), it finds the "μ" and "σ" values that fits the data first and then does transformation, while since these values have already been find according to the data it already fit upon, we just need to call transform () on the test set.

Now since we have the final X_train's and X_test, we can now see the target values (y_train and y_test) too.

```
y_train = LogDat_train_labels.copy()
y_train.head()
```

	Type of Formation_dolomite	Type of Formation_limestone	Type of Formation_sandstone	Type of Formation_sandy limestone	Type of Formation_shale	Type of Formation_shaly limestone	Type of Formation_shaly sandstone
364	0	1	0	0	0	0	0
1119	0	1	0	0	0	0	0
974	0	0	0	1	0	0	0
481	0	0	0	0	1	0	0
828	0	1	0	0	0	0	0

Figure 32: Assigning labels for training set

```
y_test_new = LogDat_test_labels.copy()
y_test_new.head()
```

	Type of Formation_dolomite	Type of Formation_limestone	Type of Formation_sandstone	Type of Formation_sandy limestone	Type of Formation_shale	Type of Formation_shaly limestone	Type of Formation_shaly sandstone
1501	0	1	0	0	0	0	0
377	0	1	0	0	0	0	0
1025	0	0	0	0	1	0	0
819	0	1	0	0	0	0	0
1364	0	0	0	0	1	0	0

Figure 33: Assigning labels for test set

We can also check the total number of times each time a formation is true (since the data is in Boolean form).

```
y_train.sum()
```

```
Type of Formation_dolomite            96
Type of Formation_limestone          698
Type of Formation_sandstone           43
Type of Formation_sandy limestone     28
Type of Formation_shale              188
Type of Formation_shaly limestone    333
Type of Formation_shaly sandstone      4
dtype: int64
```

```
y_test_new.sum()
```

```
Type of Formation_dolomite            36
Type of Formation_limestone          258
Type of Formation_sandstone           16
Type of Formation_sandy limestone     10
Type of Formation_shale               70
Type of Formation_shaly limestone    123
Type of Formation_shaly sandstone      2
dtype: int64
```

Figure 34: Checking total labels both in training and test set

3.1.2.4 : Handling targets with text attributes

When we are dealing with text attributes, whether we have them as the target values (as in our case) or a part of the training features, it is important to convert them as representative numerical values as ML algorithms cannot train on text features. There are several ways of dealing with such features.

1- Pandas.factorize ()/ Scikit-learn's LabelEncoder () method: This method converts the string categorical features into integer categorical features easier for ML algorithms to handle.

2- Scikit-learn's OneHotEncoder (): This method is similar to the method we adopted, it converts the targets into several categories according to target labels and gives true where that class is present and false when that class is absent.

The reason why OneHotEncoder () is preferred for ML algorithms is that LabelEncoder ()/ Pandas.factorize () assumes that higher the value, the better the category i.e. if we have seven classes as in our case of date here, it assumes category "7" is better.

Initially, for our data we planned to use the OneHotEncoder () technique but this wasn't required anymore as we discussed earlier as well. Getting the Pandas dummies for balanced classes requirement changed the data to Boolean form without requiring encoding the data using OneHotEncoder ().

3.1.2.5 : Visualizing the data after cleaning

We can call the .hist () again and check what the data now looks like:

```
In [34]: LogDat_drop.hist(bins=50, figsize=(20,15))
```

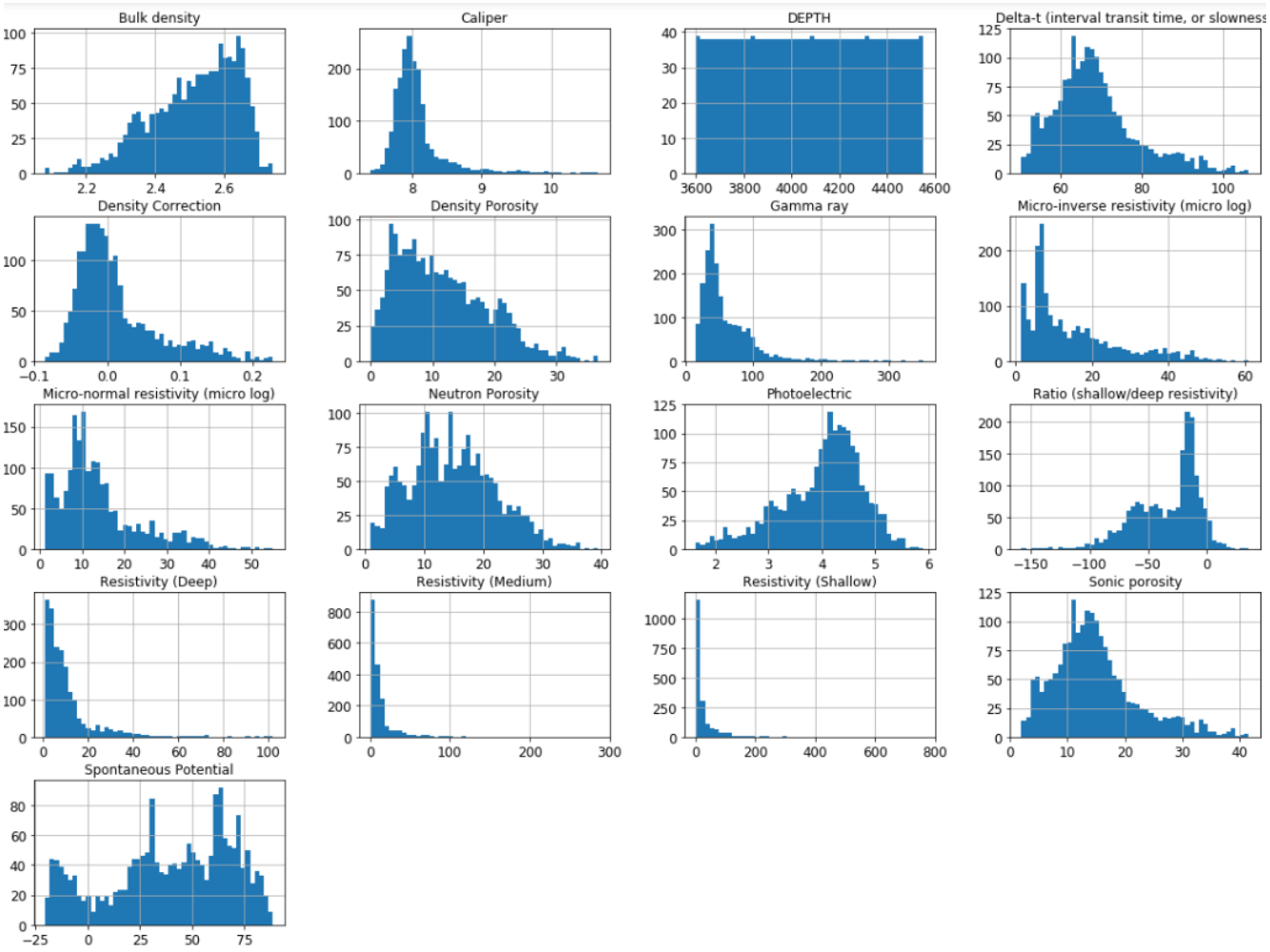

Figure 35: Histogram appearance after data cleaning

As we can, the data seems in a much better state and each feature has a certain high and low frequencies of where most of the data lies. The next step would be to plot each feature and see if they have a relation

with each and other. If some of the features have a strong relationship, we can reduce the size of the training set and make the training easier, faster and more accurate.

3.1.2.6 : Correlation matrix and Feature importances

Pandas provides a correlation matrix (.corr () method) to be used with the data frame and to check for linear relationship between each values of the features in the data frame and generates a table accordingly. A scale of -1 to 1 is generated, with positive 1 indicating a strong linearly positive relationship, i.e. if that feature is increased the corresponding linearly positive feature would also increase. 0 indicates no relation at all, while -1 would show a strong linear decrease in the corresponding linearly negative feature. Here's the code to generate this correlation matrix:

```
In [66]: corr_matrix = LogDat_train_new.corr()
         corr_matrix
```

Figure 36: Code for checking correlation matrix of data

	DEPTH	Neutron Porosity	Caliper	Density Porosity	Gamma ray	Photoelectric	Bulk density	Density Correction	Resistivity (Deep)	Resistivity (Medium)	Resistivity (Shallow)	(s
DEPTH	1.000000	0.026227	-0.160925	0.006949	-0.015251	-0.426672	-0.007264	0.288437	0.185614	0.092778	0.110453	
Neutron Porosity	0.026227	1.000000	0.346474	0.792338	0.492634	-0.609484	-0.792846	0.382943	-0.507277	-0.454351	-0.485645	
Caliper	-0.160925	0.346474	1.000000	0.210884	0.508586	-0.154071	-0.210553	0.407552	-0.099120	-0.068459	-0.120676	
Density Porosity	0.006949	0.792338	0.210884	1.000000	0.350939	-0.719724	-0.999907	0.362429	-0.366280	-0.330421	-0.362159	
Gamma ray	-0.015251	0.492634	0.508586	0.350939	1.000000	-0.289496	-0.350993	0.378688	-0.190353	-0.181184	-0.180270	
Photoelectric	-0.426672	-0.609484	-0.154071	-0.719724	-0.289496	1.000000	0.720067	-0.587235	0.331234	0.337320	0.364613	
Bulk density	-0.007264	-0.792846	-0.210553	-0.999907	-0.350993	0.720067	1.000000	-0.362460	0.367284	0.333102	0.365009	
Density Correction	0.288437	0.382943	0.407552	0.362429	0.378688	-0.587235	-0.362460	1.000000	-0.254869	-0.245138	-0.203270	
Resistivity (Deep)	0.185614	-0.507277	-0.099120	-0.366280	-0.190353	0.331234	0.367284	-0.254869	1.000000	0.864562	0.579356	
Resistivity (Medium)	0.092778	-0.454351	-0.068459	-0.330421	-0.181184	0.337320	0.333102	-0.245138	0.864562	1.000000	0.666370	
Resistivity (Shallow)	0.110453	-0.485645	-0.120676	-0.362159	-0.180270	0.364613	0.365009	-0.203270	0.579356	0.666370	1.000000	
Ratio (shallow/deep resistivity)	0.309388	0.305985	0.351541	0.189555	0.336097	-0.408454	-0.190888	0.262263	0.057239	-0.080670	-0.471655	
Spontaneous Potential	0.518315	-0.019830	0.373152	-0.208932	0.413603	-0.148010	0.208345	0.314356	0.055853	-0.027903	0.013215	
Micro-inverse resistivity (micro log)	0.000968	-0.764768	-0.151633	-0.661448	-0.236857	0.592701	0.662540	-0.373911	0.601999	0.569596	0.616441	
Micro-normal resistivity (micro log)	-0.014199	-0.779691	-0.258649	-0.632128	-0.352367	0.617986	0.633809	-0.414038	0.607740	0.584128	0.695370	
Delta-t (interval transit time, or slowness)	0.086516	0.865844	0.481840	0.726317	0.692064	-0.662070	-0.726637	0.537983	-0.486506	-0.442307	-0.449752	
Sonic porosity	0.086517	0.865844	0.481840	0.726317	0.692064	-0.662070	-0.726636	0.537983	-0.486506	-0.442307	-0.449752	

Figure 37: Correlation matrix of the data

Since it's difficult to look for each feature individually in number, we can use matplotlib to do the job of visualizing the relationships between each feature by combining Matplotlib with our modified function from Taspinar, 2018 [3].

```
In [67]: def display_corr_with_col(df, col):
             correlation_matrix = LogDat_train_new.corr()
             correlation_type = correlation_matrix[col].copy()
             abs_correlation_type = correlation_type.apply(lambda x: abs(x))
             desc_corr_values = abs_correlation_type.sort_values(ascending=False)
             y_values = list(desc_corr_values.values)[1:]
             x_values = range(0,len(y_values))
             xlabels = list(desc_corr_values.keys())[1:]
             fig, ax = plt.subplots(figsize=(10,8))
             ax.bar(x_values, y_values)
             ax.set_title('The correlation of all features with {}'.format(col), fontsize=13)
             ax.set_ylabel('Pearson correlation coefficient [abs waarde]', fontsize=13)
             plt.xticks(x_values, xlabels, rotation='vertical')
             plt.show()

         display_corr_with_col(LogDat_train_new, 'Neutron Porosity')
```

Figure 38: Modified code for displaying correlation matrix for our data (Taspnir, 2018) [3]

When we call our function with a particular feature in our training set as in the last line of the code example above, we get the following bar graph:

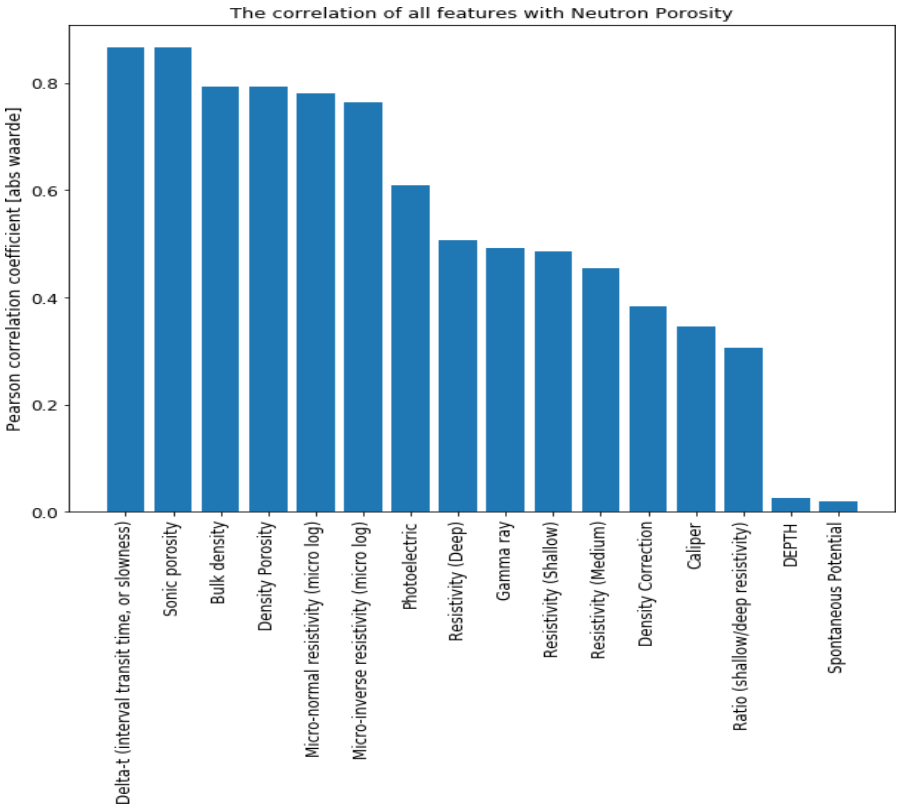

Figure 39: Correlation matrix for Neutron Porosity

As can be seen that there is a strong linear relationship between delta-t and sonic porosity with neutron porosity. Also, if we plot the same bar graph of sonic porosity, we would find that delta-t has a perfectly linear relationship with sonic porosity and similarly strong relationship with neutron porosity.

```
In [71]: display_corr_with_col(LogDat_train_new, 'Sonic porosity')
```

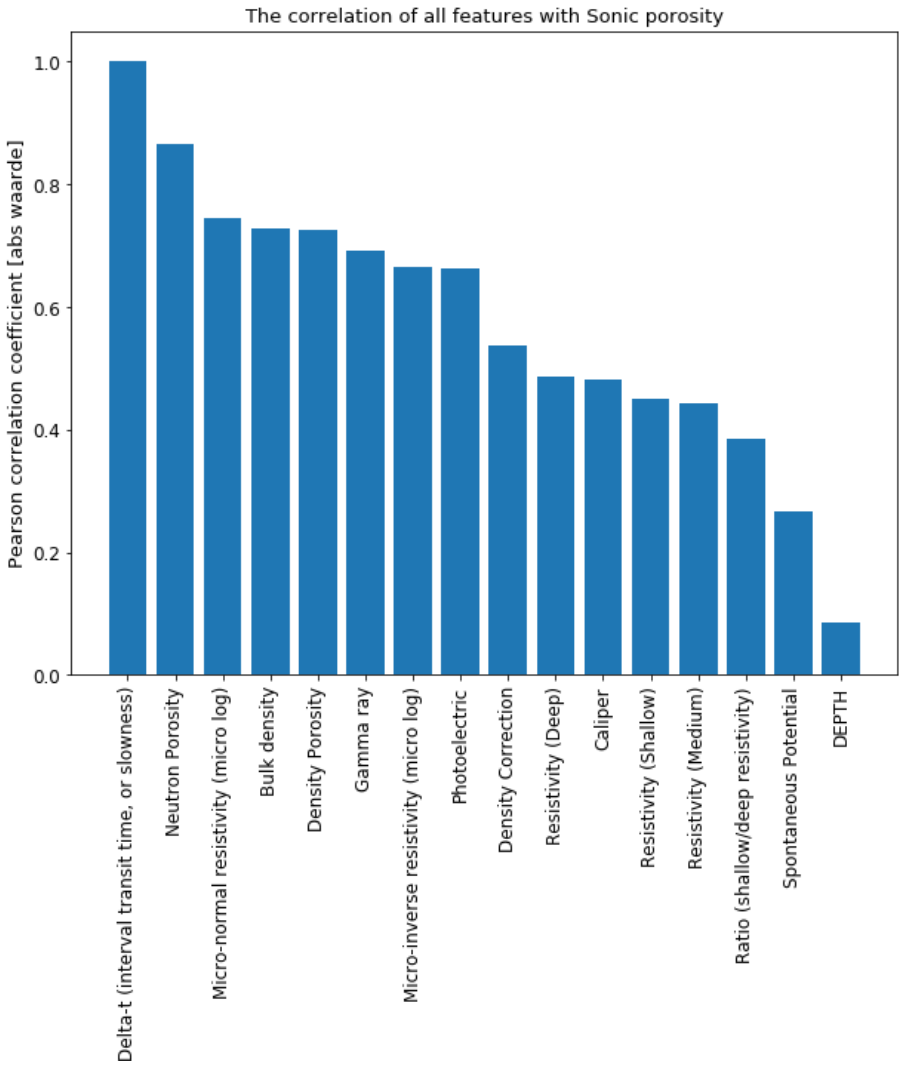

Figure 40: Correlation matrix for sonic porosity

The reason for this perfectly linear relationship of delta-t and sonic is simply because sonic porosity is calculated from delta-t and either of these features can be eliminated in training to make the training easier.

An interestingly strong relationship between neutron and sonic porosity is also observed in the case of the data of our type. This puts into question whether sonic log porosity and neutron log porosity should both be used or just one log would be enough to do the job of evaluation of porosity. This can be tested on more datasets and evaluated and save a lot of money by eliminating the use of one of these logs. Also, it can also help us in training the data with more ease.

We can see more plots for each feature individually this way and see for linear relationship and evaluate how useful they might be for this type of evaluation.

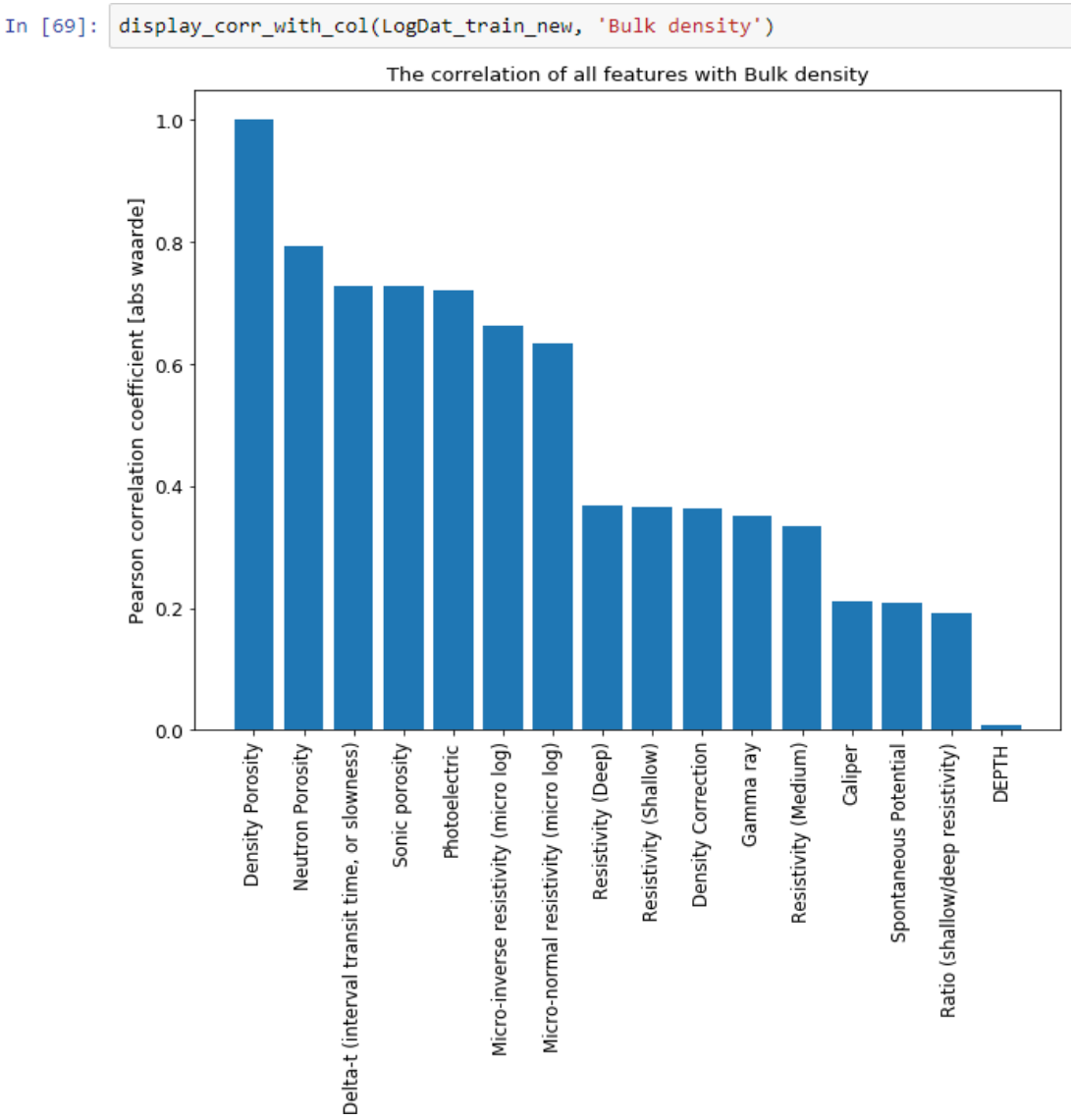

Figure 41: Correlation matrix for Bulk density

As expected, density porosity and bulk densities have a perfectly linear relationship as density porosity is calculated from bulk density. And again, neutron porosity has a strong linear relationship with bulk density/density porosity.

```
In [70]: display_corr_with_col(LogDat_train_new, 'Caliper')
```

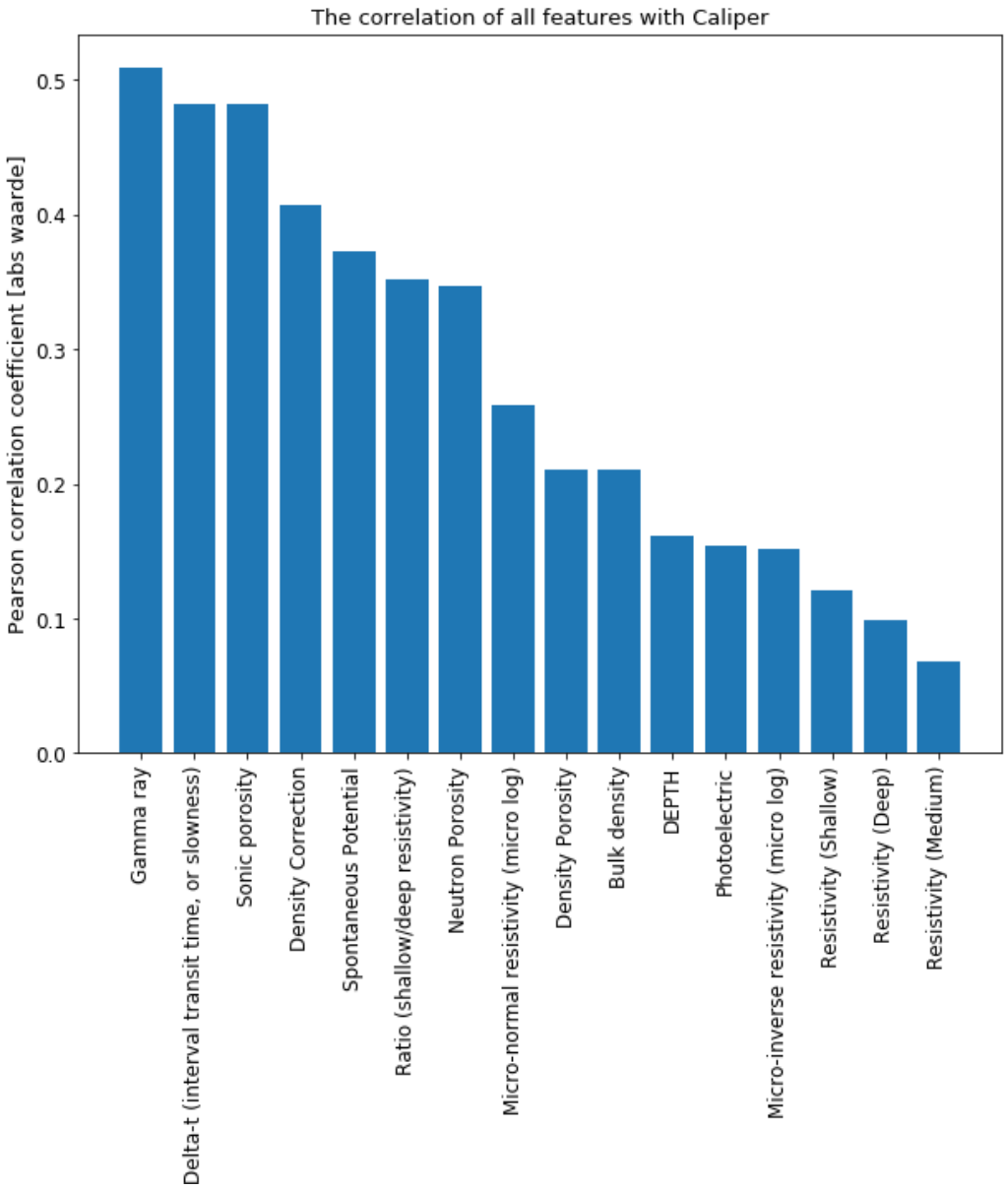

Figure 42: Correlation matrix for Caliper log

A not so strong correlation of caliper is seen with the rest of the features.

```
In [73]: display_corr_with_col(LogDat_train_new, 'Micro-normal resistivity (micro log)')
```

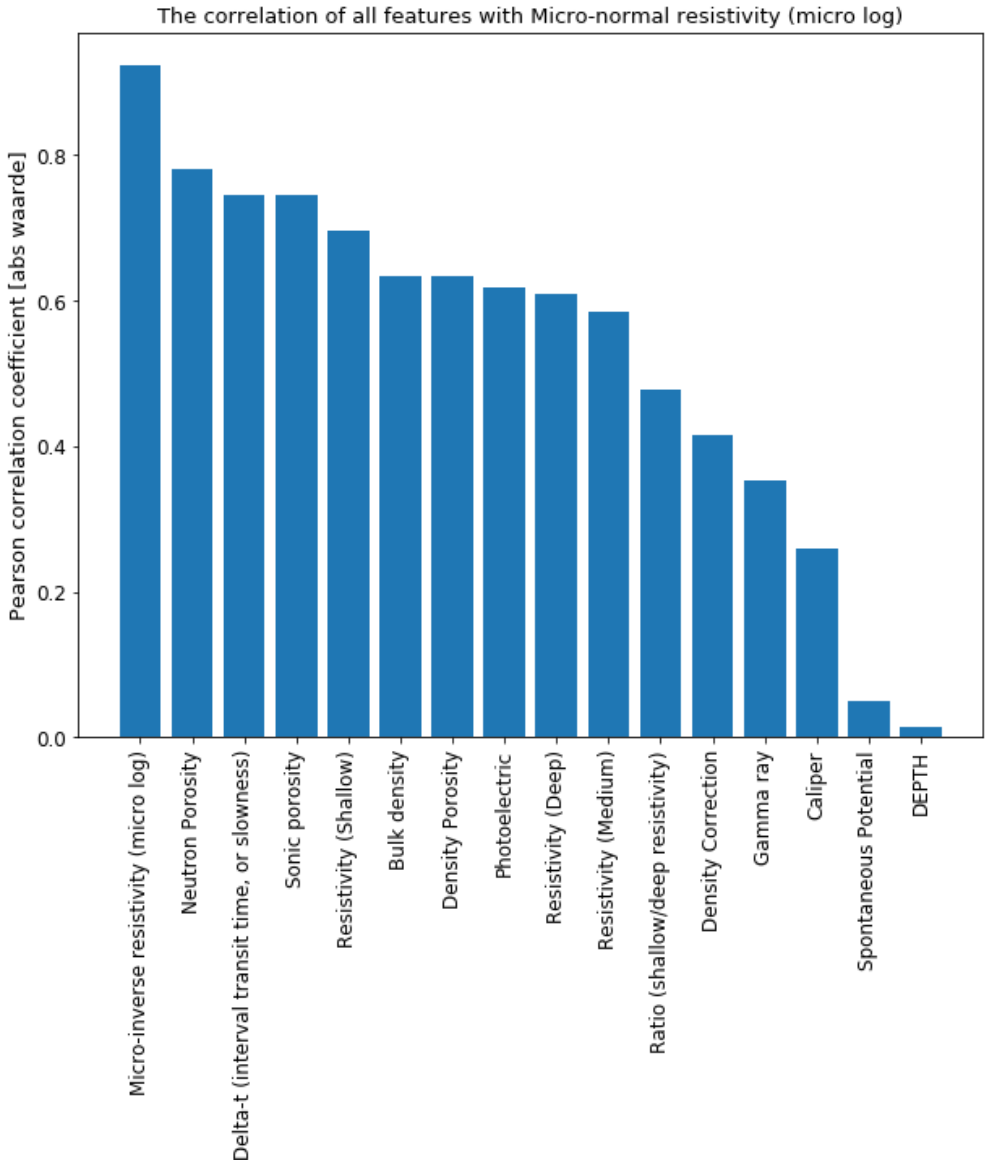

Figure 43: Correlation matrix for Micro (micro-normal) resistivity log

Again, micro-normal resistivity has a strong relationship with micro-inverse resistivity since micro-inverse resistivity is calculated from micro-normal resistivity. Both are basically used in evaluation of permeable zones across which mud cake is formed. This might help us eliminate either of the two logs again, most probably micro-inverse would be best to remove since it is calculated from micro-normal resistivity after all.

```
In [79]: display_corr_with_col(LogDat_train_new, 'Resistivity (Medium)')
```

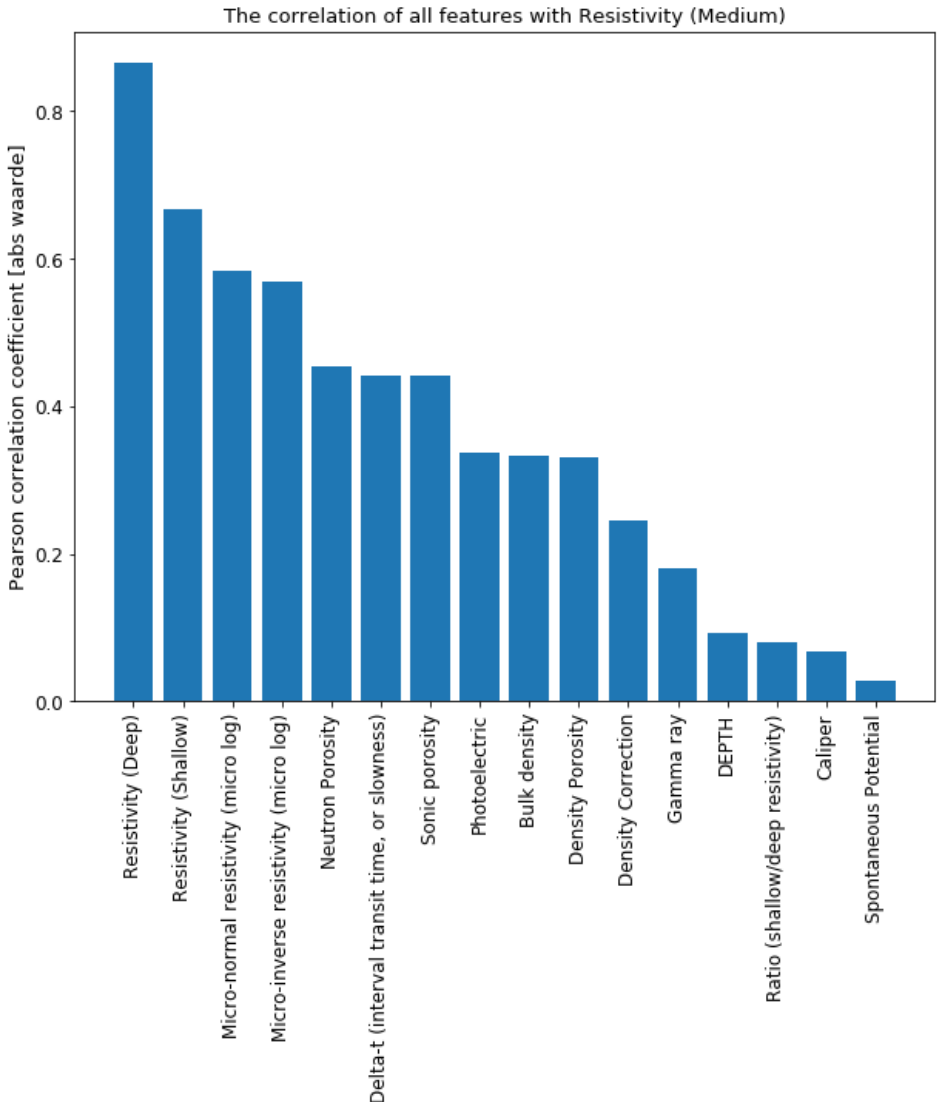

Figure 44: Correlation matrix for Resistivity (medium) log

```
In [74]: display_corr_with_col(LogDat_train_new, 'Resistivity (Shallow)')
```

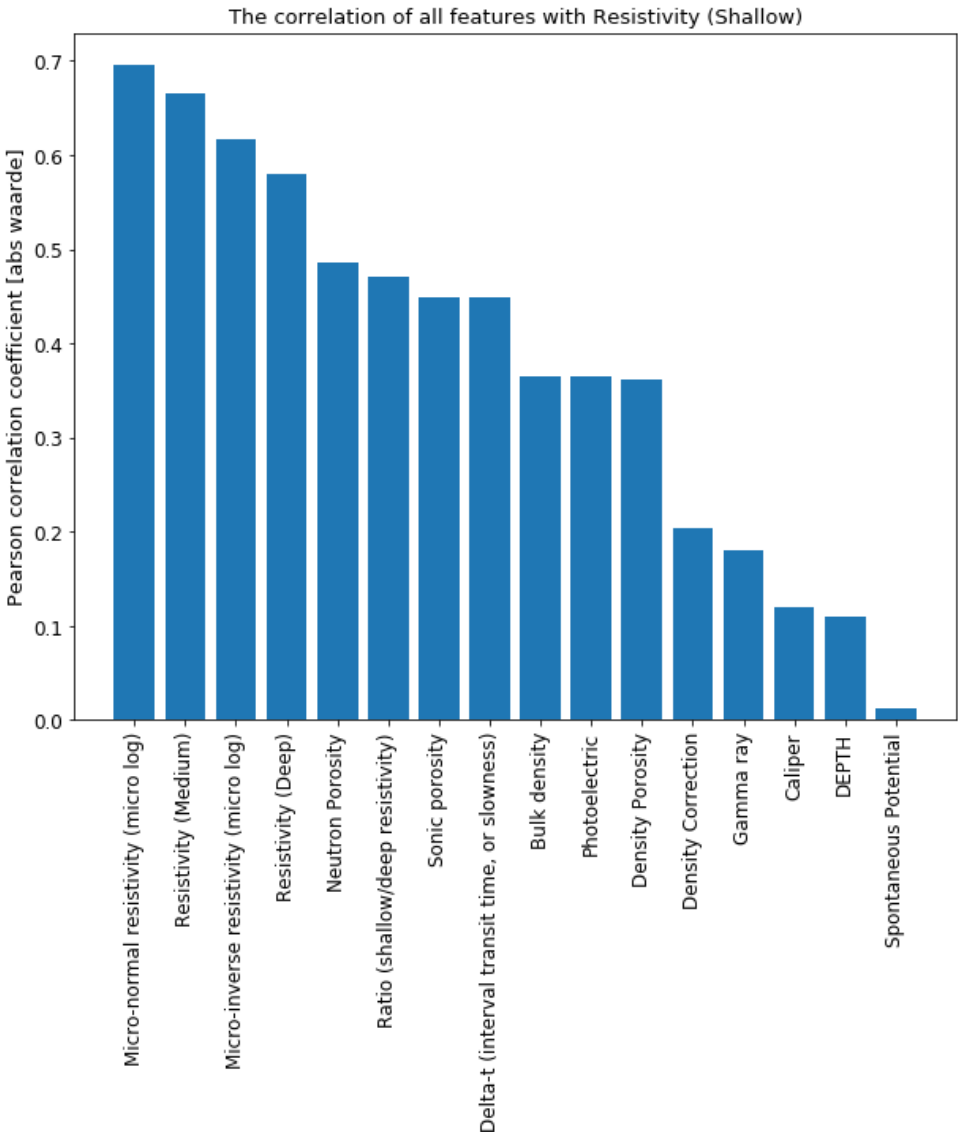

Figure 45: Correlation matrix for Resistivity (shallow) log

```
In [75]: display_corr_with_col(LogDat_train_new, 'Ratio (shallow/deep resistivity)')
```

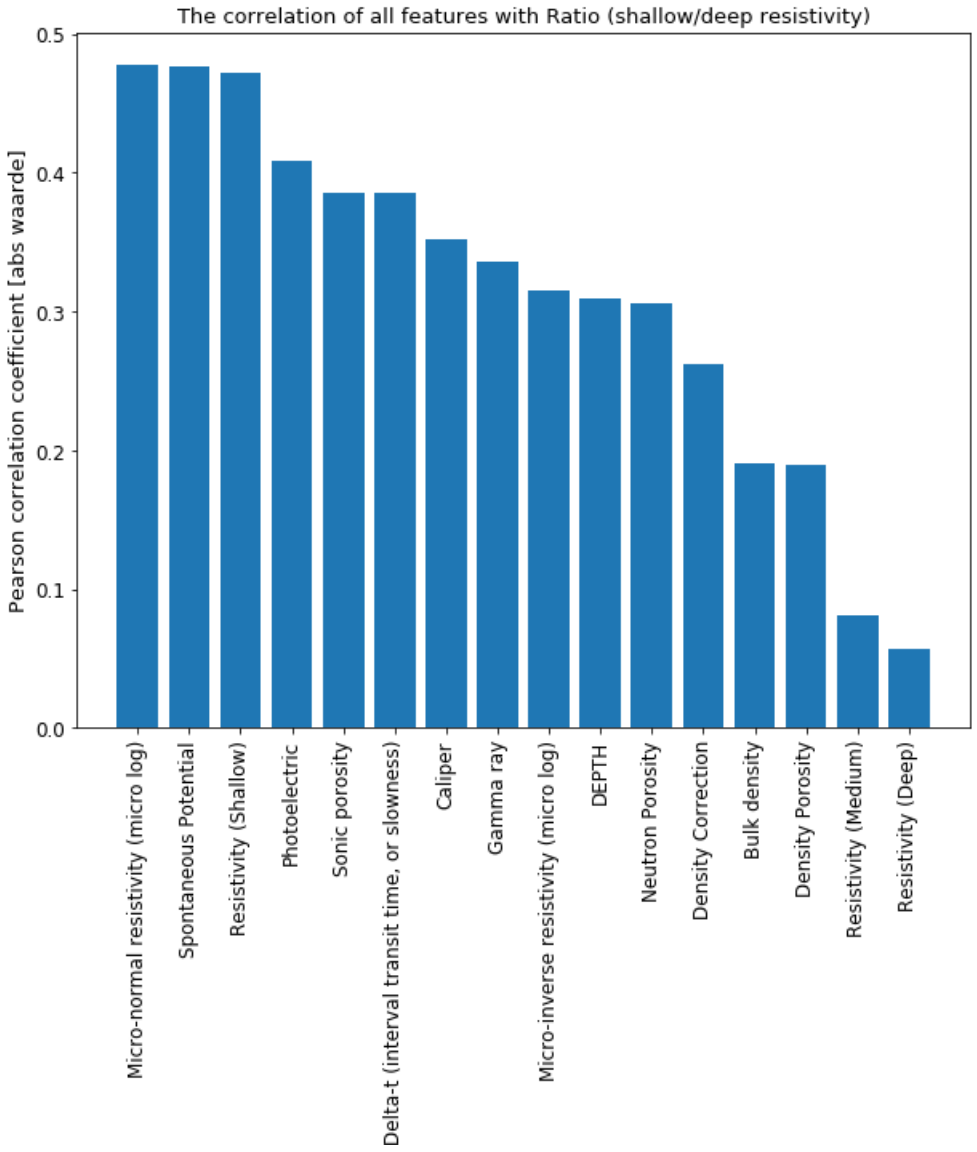

Figure 46: Correlation matrix for ratios of shallow and deep resistivity

```
In [76]: display_corr_with_col(LogDat_train_new, 'Gamma ray')
```

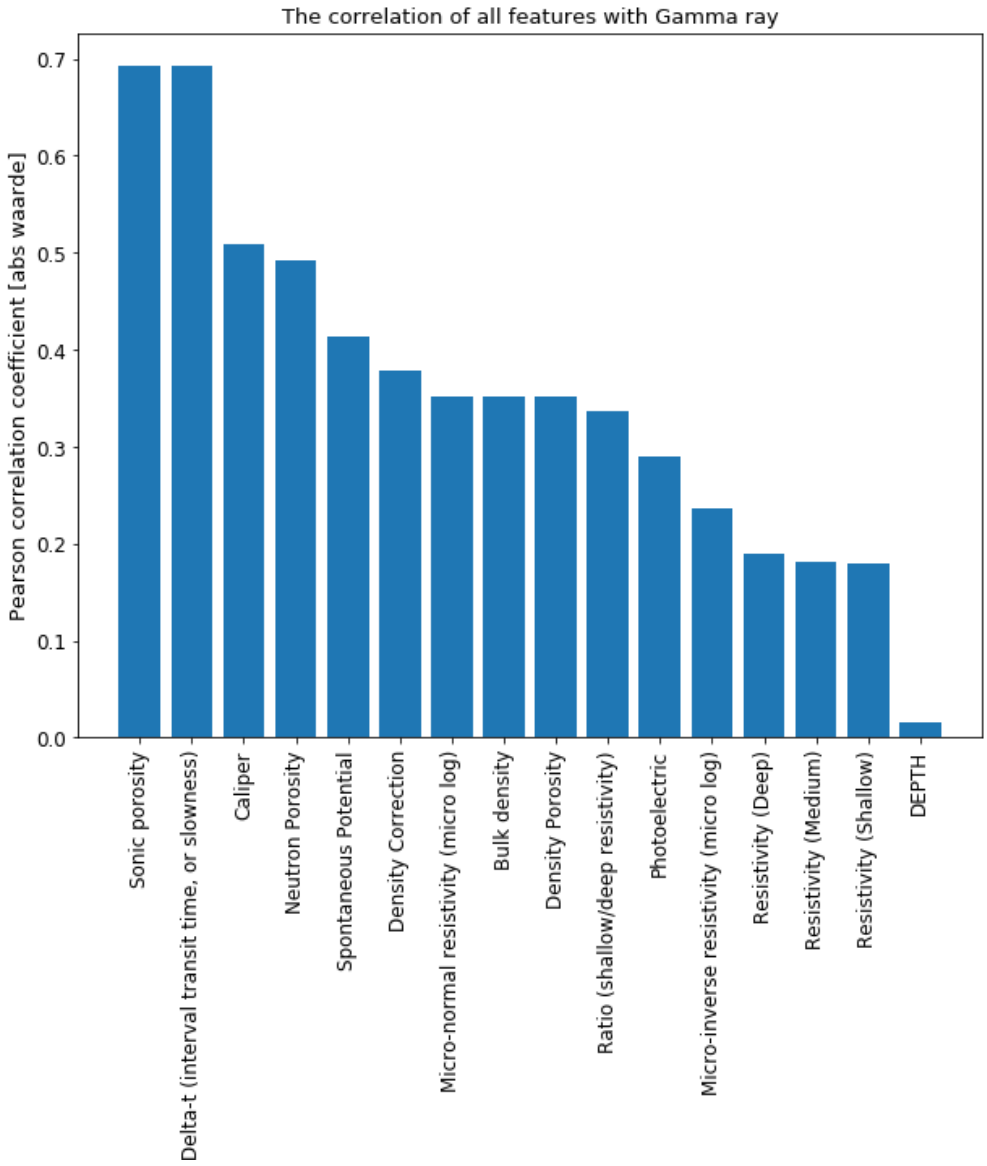

Figure 47: Correlation matrix for Gamma ray

```
In [77]: display_corr_with_col(LogDat_train_new, 'Spontaneous Potential')
```

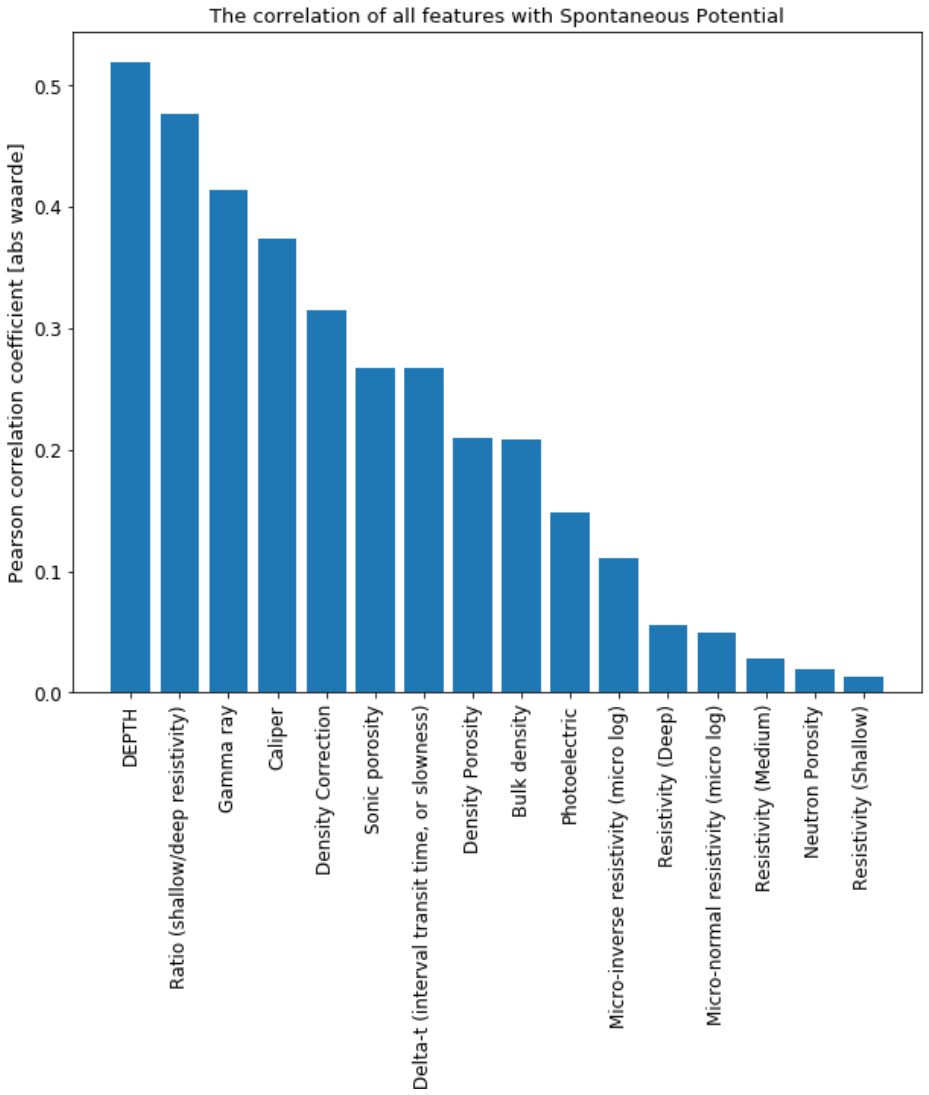

Figure 48: Correlation matrix for SP log

```
In [78]: display_corr_with_col(LogDat_train_new, 'Photoelectric')
```

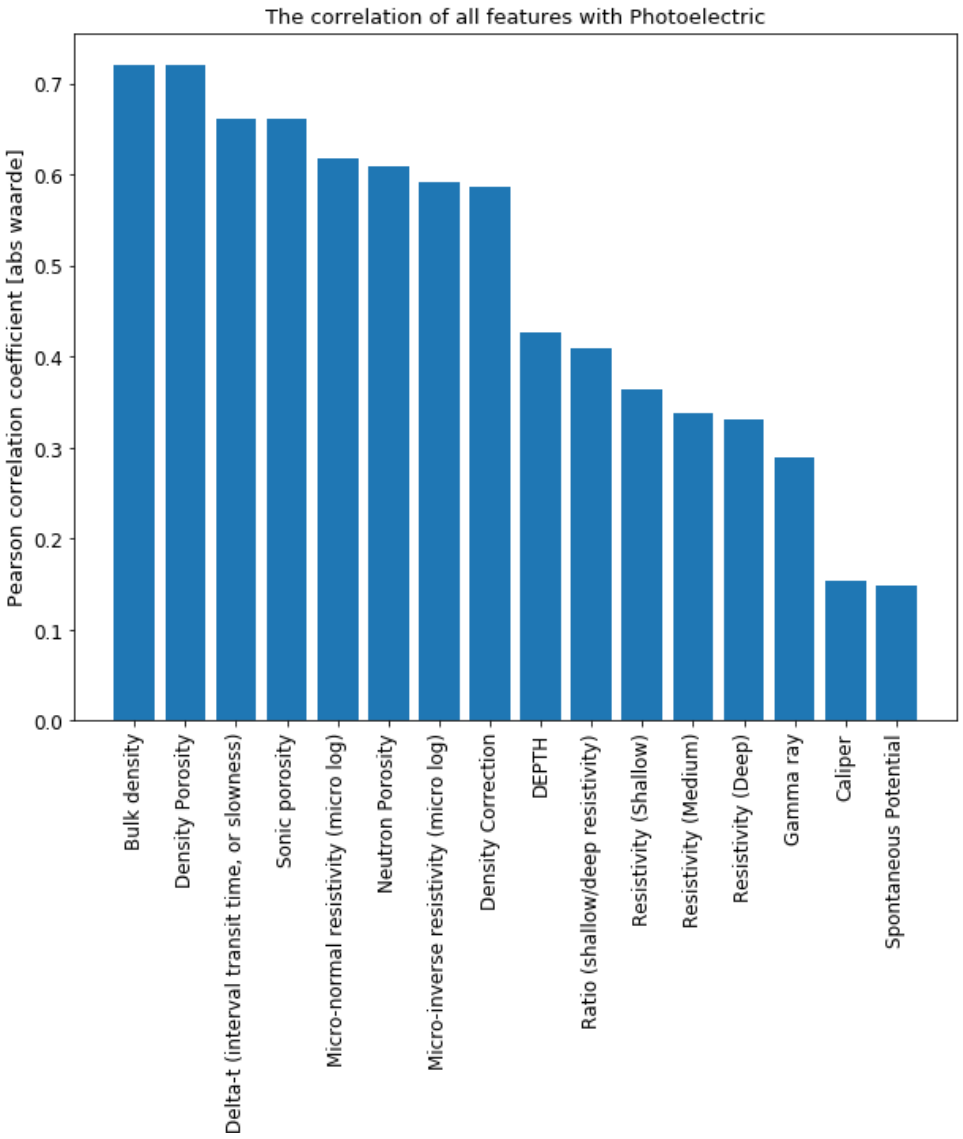

Figure 49: Correlation matrix for PE log

```
In [79]: display_corr_with_col(LogDat_train_new, 'Resistivity (Medium)')
```

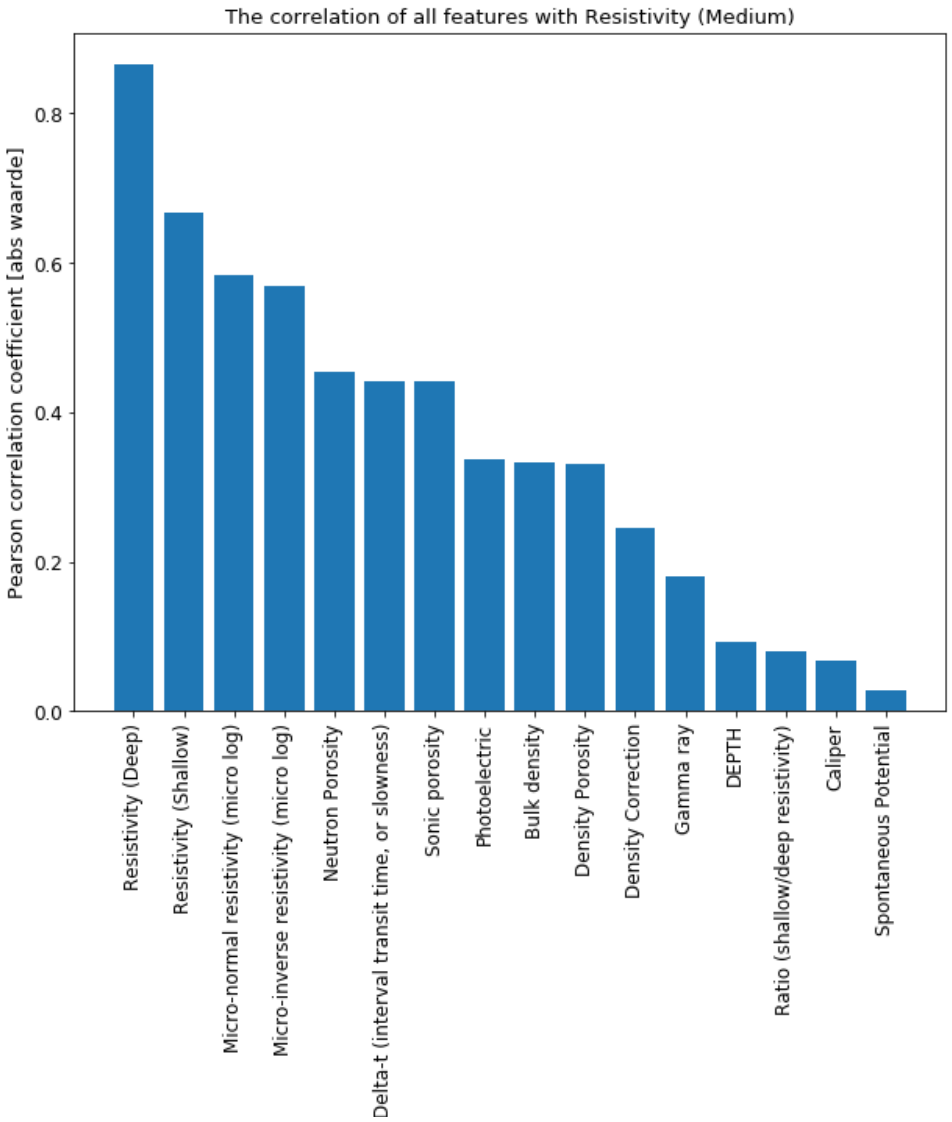

Figure 50: Correlation matrix for Resistivity (medium) log

```
In [80]: display_corr_with_col(LogDat_train_new, 'Density Correction')
```

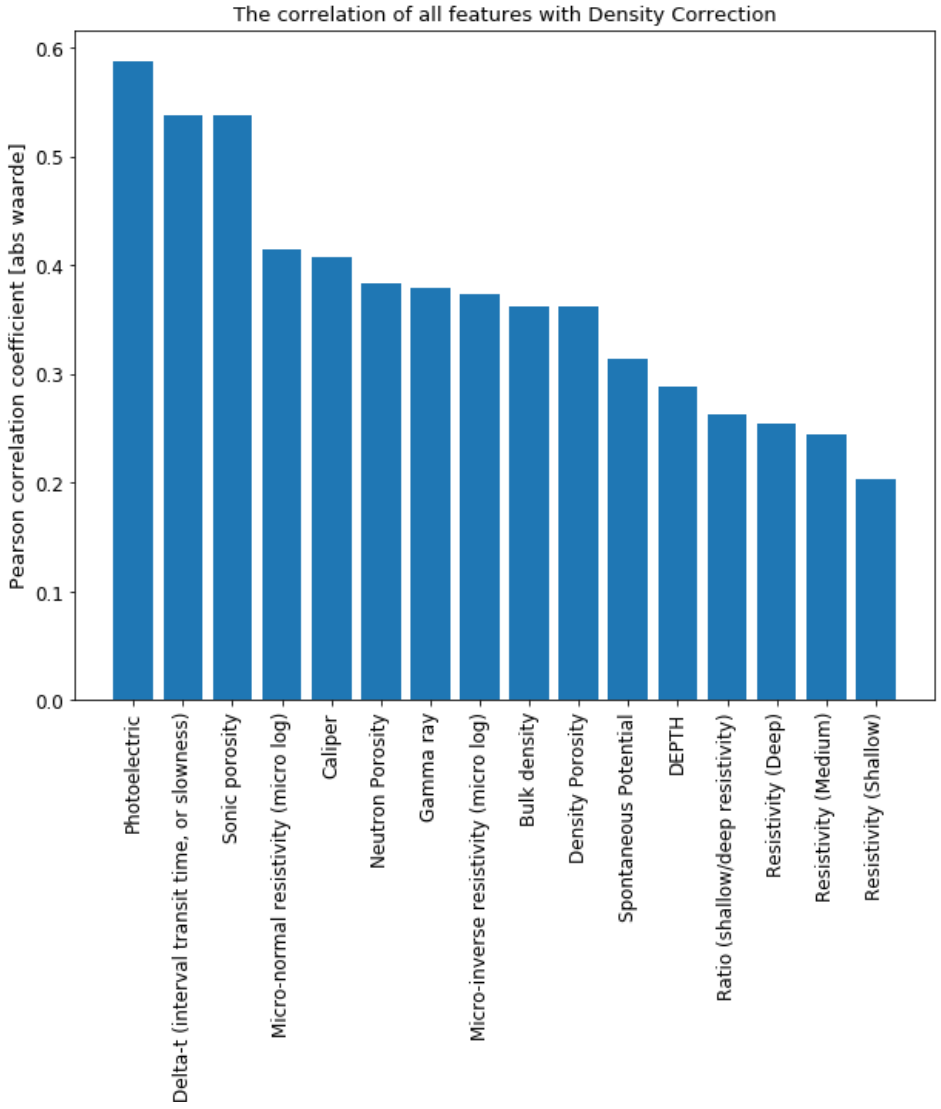

Figure 51: Correlation matrix for Density correction

With this code, we have seen some interesting relationships between many logs with each other and there is a high possibility it can be useful in eliminating the use of some of the logs due to their perfectly linear relationship and also can be also helpful in dimensionality reduction that would eliminate use of some of the features during training and make training faster, easier and more precise. But since here our actual goal was not to go towards dimensionality reduction, this can be a good area of research in future ML implementations specifically dedicated to this section only.

Another way for this subject of dimensionality reduction is by use of "*feature importances*". When we do grid search (you will learn about this just in a bit), once we fit the classifier in the grid search and fit the data with classifier in it, we can extract information about how important each feature is in training that class. If the values are below a certain range, we can drop those particular features. The implementation is shown here:

```
In [302]: feature_importances_for = grid_for_sl.best_estimator_.feature_importances_

          [ 32.69867681   2.71469709   4.32502829   2.19284759  11.43711267
            3.68548527   2.14716071   5.29196101   4.69972334   3.83732879
            2.79130235   2.46672042   7.02566772   3.58726668   3.87697245
            3.56675151   3.6552973 ] %
```

```
In [305]: attributes = ["Depth", "Neutron Porosity", "Caliper", "Density Porosity", "Gamma Ray", "Photoelctric",
                        "Bulk Density", "Density Correction", "Resistivity (Deep)", "Resistivity (Medium)", "Resistivity (Shallow)",
                        "Ratio(Shallow/Deep resistivity)", "SP", "Micro-inverse (resistivity) micro-log",
                         "Micro-normal (resistivity) micro-log", "Delta-t (transit time)", "Sonic Porosity"]
          sorted(zip(feature_importances_for*100, attributes), reverse=True)

Out[305]: [(32.698676805752939, 'Depth'),
           (11.437112671467331, 'Gamma Ray'),
           (7.0256677233376346, 'SP'),
           (5.2919610063076963, 'Density Correction'),
           (4.69972333501756, 'Resistivity (Deep)'),
           (4.3250282898616081, 'Caliper'),
           (3.876972448849632, 'Micro-normal (resistivity) micro-log'),
           (3.8373287915693957, 'Resistivity (Medium)'),
           (3.6854852718774076, 'Photoelctric'),
           (3.6552972972995796, 'Sonic Porosity'),
           (3.5872666770164936, 'Micro-inverse (resistivity) micro-log'),
           (3.5667515140079744, 'Delta-t (transit time)'),
           (2.7913023534006003, 'Resistivity (Shallow)'),
           (2.714697087901786, 'Neutron Porosity'),
           (2.4667204236786993, 'Ratio(Shallow/Deep resistivity)'),
           (2.1928475922610642, 'Density Porosity'),
           (2.1471607103925967, 'Bulk Density')]
```

Figure 52: Feature importance of Shaly-Limestone formation

In the code that is shown above, we basically did grid search on the shaly limestone class (used to find best combination of hyperparameters that would give the best cross validation score with that classifier), once we fit the grid search with the data and extract the feature relevance to training the data with the classifier (in this case Random Forest classifier was used which is discussed later). As we can see that Depth is the most important feature to train the classifier, followed by others. In this case, the bulk density doesn't seem so important to train the shaly limestone class and wouldn't be much useful if we even decided to not run a density log in this case (only 2.2% important to formation evaluation).

This way, we can evaluate each log individually that how important it can be to that particular type of formation, and we decided to drop a particular log when we know we would be dealing these kinds of formations we could save millions of dollars to the company by dropping to run some of the logs that are not so important for formation evaluation.

Similarly, we can test this on the rest of the formations too as shown below.

Feature importances of limestone:

```
grid_for_lim = GridSearchCV(for_clf_fi,param_grid=param_grid_for,cv=5, n_jobs =-1)
grid_for_lim.fit(X_train, y_train_lim)
feature_importances_lim = grid_for_lim.best_estimator_.feature_importances_
sorted(zip(feature_importances_lim*100, attributes_all), reverse=True)
```

```
[(20.490873610563774, 'Depth'),
 (12.09548368712561, 'Photoelectric'),
 (7.5876354108716608, 'Gamma Ray'),
 (7.199221662111591, 'Sonic Porosity'),
 (5.9924574197004841, 'SP'),
 (5.6892619027191973, 'Density Correction'),
 (5.4727581603199775, 'Delta-t (transit time)'),
 (4.037953279469467, 'Neutron Porosity'),
 (3.9772708914231534, 'Resistivity (Deep)'),
 (3.9157159352337554, 'Bulk Density'),
 (3.8785590650677042, 'Resistivity (Medium)'),
 (3.8378661049938105, 'Density Porosity'),
 (3.8082657018084038, 'Micro-inverse (resistivity) micro-log'),
 (3.4521001554844304, 'Micro-normal (resistivity) micro-log'),
 (3.2643772249487251, 'Caliper '),
 (2.6545476810977231, 'Resistivity (Shallow)'),
 (2.645652107060513, 'Ratio(Shallow/Deep resistivity)')]
```

Figure 53: Feature importance of Limestone

Feature importances of shale:

```
grid_for_shale = GridSearchCV(for_clf_fi,param_grid=param_grid_for,cv=5, n_jobs =-1)
grid_for_shale.fit(X_train, y_train_shale)
feature_importances_shale = grid_for_shale.best_estimator_.feature_importances_
sorted(zip(feature_importances_shale*100, attributes_all), reverse=True)
```

```
[(20.950655660649275, 'Gamma Ray'),
 (8.7649983045863511, 'Depth'),
 (7.8745311068874013, 'SP'),
 (7.544206710447952, 'Sonic Porosity'),
 (6.632776920956335, 'Delta-t (transit time)'),
 (6.1290876680814366, 'Density Correction'),
 (5.1297756155294962, 'Caliper '),
 (4.7669950717085996, 'Micro-normal (resistivity) micro-log'),
 (4.6914561836071549, 'Resistivity (Deep)'),
 (4.6769052063688381, 'Resistivity (Medium)'),
 (3.9460133944234212, 'Micro-inverse (resistivity) micro-log'),
 (3.417788758909404, 'Ratio(Shallow/Deep resistivity)'),
 (3.2810824051178686, 'Photoelectric'),
 (3.2502028490509072, 'Neutron Porosity'),
 (3.1743094666705343, 'Resistivity (Shallow)'),
 (2.8865801898158385, 'Density Porosity'),
 (2.8826344871891747, 'Bulk Density')]
```

Figure 54: Feature importance of Shale

Feature importances of Sandy-Limestone:

```
grid_for_sandlim = GridSearchCV(for_clf_fi,param_grid=param_grid_for,cv=5, n_jobs =-1)
grid_for_sandlim.fit(X_train, y_train_sandlim)
feature_importances_sandlim = grid_for_sandlim.best_estimator_.feature_importances_
sorted(zip(feature_importances_sandlim*100, attributes_all), reverse=True)
```

```
[(14.707880882714836, 'Resistivity (Shallow)'),
 (13.364730255845391, 'Bulk Density'),
 (11.461630168056711, 'Resistivity (Deep)'),
 (11.340180227209865, 'Density Porosity'),
 (9.2415438757067445, 'Resistivity (Medium)'),
 (8.8391414847332239, 'Neutron Porosity'),
 (5.7003725884107324, 'Sonic Porosity'),
 (5.5837958853639584, 'Delta-t (transit time)'),
 (4.1046578990570186, 'Micro-normal (resistivity) micro-log'),
 (3.2183081956192647, 'SP'),
 (2.6933720135368087, 'Depth'),
 (2.4151182886140723, 'Micro-inverse (resistivity) micro-log'),
 (2.3275541139553777, 'Ratio(Shallow/Deep resistivity)'),
 (1.9521452493236229, 'Photoelectric'),
 (1.7586431580477573, 'Gamma Ray'),
 (0.94090848517096937, 'Density Correction'),
 (0.35001722863365575, 'Caliper ')]
```

Figure 55: Feature importance of Sandy-Limestone

Feature importances of Shaly Sandstone:

```
grid_for_ss = GridSearchCV(for_clf_fi,param_grid=param_grid_for,cv=4, n_jobs =-1)
grid_for_ss.fit(X_train, y_train_ss)
feature_importances_ss = grid_for_ss.best_estimator_.feature_importances_
sorted(zip(feature_importances_ss*100, attributes_all), reverse=True)
```

```
[(21.596809649545197, 'Depth'),
 (15.914938857340951, 'Photoelectric'),
 (7.1646710110631329, 'Bulk Density'),
 (7.136841528247909, 'SP'),
 (6.1458162965539538, 'Density Correction'),
 (5.9321190717924184, 'Density Porosity'),
 (5.7715629320794433, 'Ratio(Shallow/Deep resistivity)'),
 (5.0460116670441417, 'Gamma Ray'),
 (4.0500786800342006, 'Caliper '),
 (3.2224969769496266, 'Micro-normal (resistivity) micro-log'),
 (3.1413058863587016, 'Resistivity (Deep)'),
 (3.0305749558090955, 'Neutron Porosity'),
 (2.8731256647119627, 'Resistivity (Medium)'),
 (2.5257474251301901, 'Sonic Porosity'),
 (1.9758618619006039, 'Delta-t (transit time)'),
 (1.5080152949014241, 'Micro-inverse (resistivity) micro-log'),
 (1.4640222405370318, 'Resistivity (Shallow)')]
```

Figure 56: Feature importance of Shaly-Sandstone

Feature importances of Dolomite:

```
grid_for_dol = GridSearchCV(for_clf_fi,param_grid=param_grid_for,cv=5, n_jobs =-1)
grid_for_dol.fit(X_train, y_train_dol)
feature_importances_dol = grid_for_dol.best_estimator_.feature_importances_
sorted(zip(feature_importances_dol*100, attributes_all), reverse=True)
```

```
[(53.509366629544509, 'Depth'),
 (11.949567088220114, 'Caliper '),
 (6.1893849633695854, 'SP'),
 (5.9758459514743212, 'Gamma Ray'),
 (3.9081768771959293, 'Photoelectric'),
 (3.5748245958727347, 'Neutron Porosity'),
 (2.0693306345007256, 'Resistivity (Deep)'),
 (2.0206560970330965, 'Ratio(Shallow/Deep resistivity)'),
 (1.9364464930680714, 'Density Porosity'),
 (1.8154415745453487, 'Bulk Density'),
 (1.6393981485793421, 'Density Correction'),
 (1.5617315647734751, 'Resistivity (Medium)'),
 (1.1101369349903425, 'Delta-t (transit time)'),
 (0.7678410807304582, 'Resistivity (Shallow)'),
 (0.75319282785050801, 'Sonic Porosity'),
 (0.68842656439892325, 'Micro-normal (resistivity) micro-log'),
 (0.53023197385250531, 'Micro-inverse (resistivity) micro-log')]
```

Figure 57: Feature importance of Dolomite

Feature importances of Sandstone:

```
grid_for_sand = GridSearchCV(for_clf_fi,param_grid=param_grid_for,cv=5, n_jobs =-1)
grid_for_sand.fit(X_train, y_train_sand)
feature_importances_sand = grid_for_sand.best_estimator_.feature_importances_
sorted(zip(feature_importances_sand*100, attributes_all), reverse=True)
```

```
[(36.309046019421423, 'Photoelectric'),
 (16.988444654943709, 'Depth'),
 (10.625009294149367, 'Gamma Ray'),
 (7.0372921222701041, 'Bulk Density'),
 (6.2082456104717005, 'Neutron Porosity'),
 (6.1994382438241598, 'Density Porosity'),
 (2.3100252234031458, 'Caliper '),
 (2.1072911414102631, 'Delta-t (transit time)'),
 (1.9954540514952033, 'Sonic Porosity'),
 (1.994056257803549, 'Resistivity (Deep)'),
 (1.8722947450454466, 'SP'),
 (1.5215307078817599, 'Density Correction'),
 (1.1586896609418966, 'Resistivity (Medium)'),
 (1.0650840258643131, 'Resistivity (Shallow)'),
 (0.98184477645290591, 'Ratio(Shallow/Deep resistivity)'),
 (0.92377160122838375, 'Micro-normal (resistivity) micro-log'),
 (0.70248186339264063, 'Micro-inverse (resistivity) micro-log')]
```

Figure 58: Feature importance of Sandstone

There are however, some limitation to this method. First of all, we can see that there are some “absurd” relations to some features such as “Depth” being less related to the log, like in case of shaly limestone formation it is just 2.7% important. This reason for this limitation is because as we will later as well, that when the classifier is not trained enough on the instances under-fitting can occur (due to not enough training instances in small data as ours, more about which we will learn), thus, some relations cannot be actually true until tested on more data and checked in practice for research purposes on the field.

Also, not all classifiers have the feature importance parameter (such as Random Forest classifier in our case did) so we cannot confirm the accuracy of predictions by our classifier by confirmation from other classifiers that we used (such as SVM or K-Nearest Neighbor classifiers, about which more later).

Being one of the most popular and strongest ML algorithm available to date we can rely on the predictions by a classifier like Random Forest about the feature importance, but the predictions needs to be confirmed with use of more classifiers, more data and lab research.

3.1.2.6 : ML algorithms implementation on dataset

3.1.2.6.1 : *Training individual classes*

There are several ways to train the type of data which contains many classes with the target class itself. In the case of geological log data as ours, we have a set of formations in the target class "Types of Formation" including limestone, shaly limestone, shale, dolomite, sandstone, shaly sandstone and sandy limestone. This is an example of typical "Classification" based ML and we would want to train each class individually and combine them together towards the end.

As already discussed, we have converted the targets into Boolean form and can separate each class/formation and use it as a part of training targets from training set and test targets from test sets. We can use the Pandas functions to do this for each type of formation.

```
In [63]: y_train_sl = y_train["Type of Formation_shaly limestone"]
         y_train_sl.head()

Out[63]: 364     0
         1119    0
         974     0
         481     0
         828     0
         Name: Type of Formation_shaly limestone, dtype: uint8

In [64]: y_train_sl.shape

Out[64]: (1390,)

In [65]: y_test_sl = y_test_new["Type of Formation_shaly limestone"]
         y_test_sl.head()

Out[65]: 1501    0
         377     0
         1025    0
         819     0
         1364    0
         Name: Type of Formation_shaly limestone, dtype: uint8

In [66]: y_test_sl.shape

Out[66]: (515,)
```

Figure 59: Assigning training labels for individual class of Shaly Limestone

3.1.2.6.2 : *Types of ML algorithms*

There are different types of algorithms available in the Scikit-Learn's library programmed to train such classification example. We will discuss the algorithms we have used in our classification problem, their working principles and how does these algorithms work before showing its implementation. More details about all the algorithms can be found from Scikit-Learn manuals [2] and Geron, 2017.

3.1.2.6.2.1 : Logistic Regression

Logistic regression (also called *Log Regression*) is mainly used to know the probability of an instance belonging to a particular class (e.g., 50% chances of an instance belongs to shaly limestone type of formation). If the estimated probability is greater than 50%, the instance belongs to that class (or it would give a True for an answer, or labeled "1", called the positive class), otherwise it does not belong to that class (i.e., it is a negative class, labeled "0" or False). Thus, it is a useful classifier for binary classification examples as ours.

Logistic regression involves the use of mathematical models to estimate the probability which outputs a number between 0 and 1 based on another model that predicts if the instance is ≥ 50% to output 1 if true otherwise 0.

3.1.2.6.2.2 : Support Vector Machine (SVM)

There are several classifiers used for different applications. Mainly they are divided as:

(a) Linear SVM Classification
(b) Non-linear SVM Classification

As the name implies that Linear SVM classification is used for linear data and vice versa for non-linear SVM classification.

3.1.2.6.2.2.1 : Linear SVM

classifier basically draws a decision boundary by fitting the widest possible street between the classes i.e. by selecting the instances in the data located on the edge of the street (points that separate the two types of classes) it plots a decision boundary that separates the two classes. This is called *large margin classification*. The instances that is used to decide the decision boundary are called *support vectors*.

Making a decision boundary with SVM classifier involving care with the margins. This can be seen in figure below. The instances can often end up on the street (i.e. the instances can end up between the

decision boundaries of classes due to a narrow decision boundary or due to *hard margin classification* which forces the instances to be off the street but on the right side let's say).

To counter this, we need to have two things in mind, the street be as wide as possible and limiting the *margin violation* by keeping the instances off the street. This is called *soft margin classification*. Using a regularization term called "*C*" in Scikit-Learn we can control this classification. Usually smaller the "*C*" value, wider the street but larger the margin violations, and higher the "C" value, smaller the margin and many instances may end up on the street.

Also, higher "C" values may cause the model to overfit. Toying with the value "C" parameter to find just the right value can be done by GridSearchCV (discussed later). Figure below demo nstrates what is discussed above for an example.

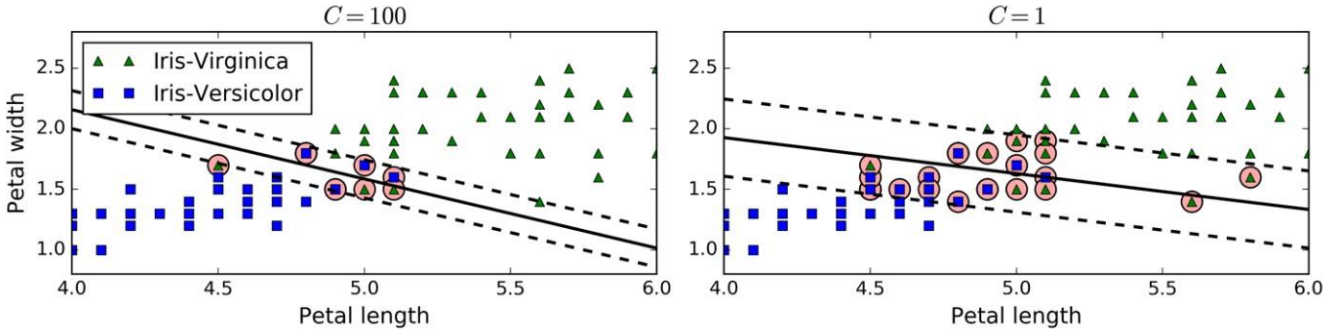

Figure 60: SVM classifier example (Geron, 2017)

3.1.2.6.2.2.2 : Nonlinear SVM

For dealing with nonlinear datasets we can add more features such as polynomial features which can result in nonlinear data converting to linearly seperable data. An example of this is shown with a figure.

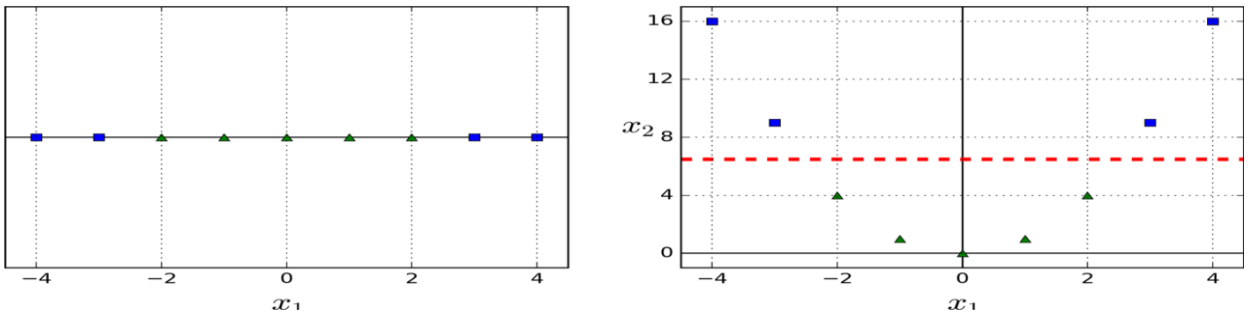

Figure 61: Non-linear SVM classification on Iris-data (Geron, 2017)

The figure shows a dataset with just one feature x_1 which as can be seen is not linearly separable, but upon adding another feature $x_2 = (x_1)^2$ (which is what adding polynomial features mean), the resulting 2D dataset is now perfectly linearly separable.

3.1.2.6.2.2.2.1: Kernel trick & similarity functions in Gaussian RBF Kernel

To tackle non-linear problems by adding features, we can compute the number of features to be added to make the data linearly separable by using the *similarity functions*. The similarity functions measures how much each instance resembles a particular *landmark.*

Let's consider the last example discussed in polynomial features to explain this. Suppose we have two *landmarks* $x_1 = -2$ and $x_1 = 1$ and we define the similarity function to be the Gaussian *Radial Basis Function* (RBF) with $\gamma = 0.3$.

Suppose an instance $x_1 = -1$, it is located at a distance of 1 from the first landmark and 2 from the second landmark. Using the mathematical relationship shown below, we are able to transform the data on the left side of the figure below to a linearly separable data on the right side of the figure.

$$\phi_\gamma(\mathbf{x}, \ell) = \exp\left(-\gamma \| \mathbf{x} - \ell \|^2\right)$$

Figure 62: Similarity functions equation, Geron (2017)

Where,

x = value of the instance

l = landmarks

$\phi\gamma$ (x, l) = similarity function

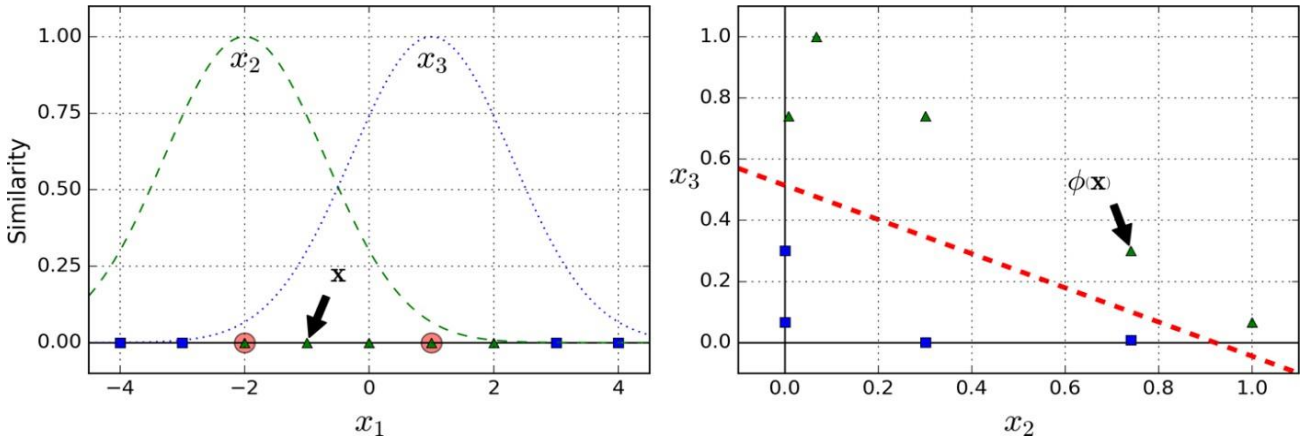

Figure 63: Similarity functions in Gaussian RBF kernel

To select the landmarks, the best way would be to create the landmark at the location of each instance in the dataset which would create many dimensions and thus, increase the chances of transformation of training set linearly separable.

Now since, it would be computationally expensive to calculate all the additional features using the similarity function, especially when dealing with large dataset, the concept of "*kernel trick"* comes into play. The kernel trick makes it possible to obtain a similar result as if we added many similarity features, without actually having to add them.

3.1.2.6.2.3 : K-Nearest Neighbors Classifier

The K-Nearest neighbors algorithm is a simple classifier that stores the instances of the training data and classifies based on the majority vote of the nearest neighbors at each point. A query is assigned in the data class that which class has the most representatives within the nearest neighbors of a selected point.

The selection of the value of nearest neighbors to look at ("*K*") is highly data-dependent. Generally, larger k values suppresses the noise, but makes the classification boundaries less distinct.

The "*weights*" function of the nearest neighbors classifier is used to assign the way weighing is done for each instance in the nearest neighbors of the selected point. If "weight" is uniform, it would mean that each neighbor of the selected point has equal weight in the majority vote of the classifier irrespective of the distance from the query/selected point. On the other hand, if "weight" is "distance", it would mean that the closest point to the query point has more weightage in the majority voting than the farthest point.

3.1.2.6.2.4 : Gaussian Naïve Bayes

The algorithm is based on Bayes theorem simply assumes that the presence of a particular feature in a class is unrelated to the presence of any other feature. Let's consider a simple example of a fruit to explain this better for which further details can be found from Vidhya, 2015. [4]

A fruit maybe be called an apple when it is red, round and about 4 inches in diameter. Although, these features depend on each other upon the existence of the other features, all of these properties independently contribute to the probability of this fruit being called an apple and this is why it called 'Naïve'.

It is an easy to build algorithm and is more suitable for large data sets. The following equation shows the Bayes theorem upon which this algorithm is based on:

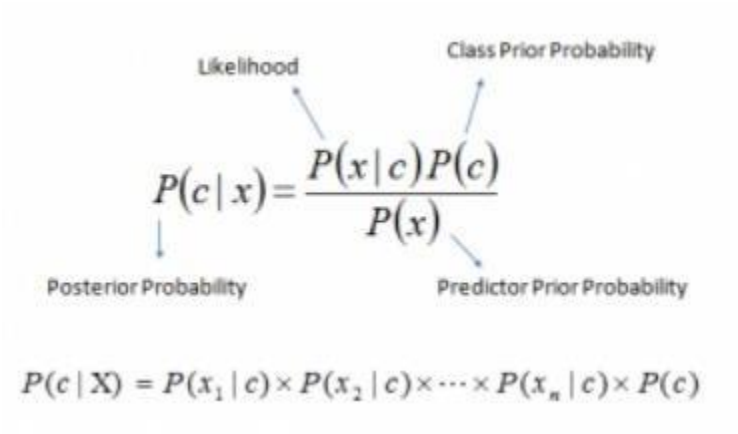

Figure 64: Working equation behind Naive Bayes algorithm [4]

Let's take a simple example of how the algorithm works. Suppose a training data with a set of weather conditions and a target variable "Play". Basically, if poor weather conditions occur, players will not "play" and vice versa.

Weather	Play
Sunny	No
Overcast	Yes
Rainy	Yes
Sunny	Yes
Sunny	Yes
Overcast	Yes
Rainy	No
Rainy	No
Sunny	Yes
Rainy	Yes
Sunny	No
Overcast	Yes
Overcast	Yes
Rainy	No

Figure 65: Extracting probabilities based on instances [4]

The dataset is first converted into a frequency table:

Frequency Table		
Weather	No	Yes
Overcast		4
Rainy	3	2
Sunny	2	3
Grand Total	5	9

Figure 66: Frequency table of instances [4]

A likelihood table is now made by finding the probabilities in the conditions when the weather is overcast.

Likelihood table				
Weather	No	Yes		
Overcast		4	=4/14	0.29
Rainy	3	2	=5/14	0.36
Sunny	2	3	=5/14	0.36
All	5	9		
	=5/14	=9/14		
	0.36	0.64		

Figure 67: Likelihood table based on instances [4]

Using the Bayes theorem shown earlier, we can calculate the posterior probability for each class. The class with the highest probability is the outcome of the prediction. In this case, the likelihood for players playing in overcast condition is 64%.

The Naïve Bayes algorithm uses a similar method to predict the probability of different classes based on different attributes and is used mostly in text classification and multiple class classification problem as in the case of our data.

3.1.2.6.2.5 : Decision Tree

To explain the Decision Tree classifier, let's consider the following example:

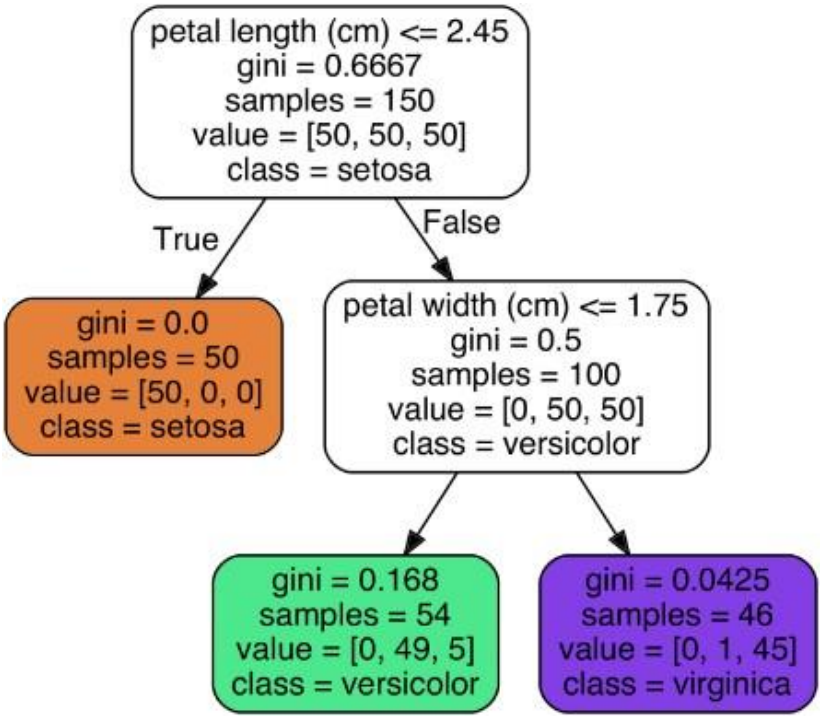

Figure 68: Classification by Decision Tree, Geron (2017)

The figure shown above is from training of an IRIS dataset which involves classifying three types of flowers Iris-setosa, Iris-versicolor & Iris-virginica. The classification is done based on only two features i.e., petal length and width. We start the root node (depth 0, at the top of the figure). This node asks the question that whether the petal length of the flower is less than or equal to 2.45 cm. If this is true, we move to the root's left child node (depth 1, left) which is a leaf node (i.e., it does not have children nodes like the node at depth 0 (top) had). Now it does not ask any more questions for this node and predicts it is Iris-setosa flower.

Similarly, if the flower would have a petal length greater than 2.45 cm it would move to the right child node and this node would be asked another question since this is not the leaf node. This time, it would ask that if the petal width is less than or equal to 1.75 cm. If true than it would go to depth 2, left node and since this is a leaf node it would predict the flower is Iris-versicolor.

On the other hand, if the flower petal width was greater than 1.75 cm it would move to the depth 2, right child node which is again a leaf node and would predict the flower to be Iris-virginica. This is how simply Decision Tree works.

A node's *"sample"* attribute counts how many training instances it applies to, for example if we had total 150 samples in the first node (depth 0), out of them 50 had a petal length less than or equal to 2.45 cm (depth 1, left node), and 100 had a petal length greater than 2.45 cm. The "*value*" attribute tells us how many training instances of each class this node applies to. For example, the left child node (depth 1) applies to 50 out of 50 Iris-setosa and none of this node is applied to other two flowers. Finally, the "*gini*" attribute is a measurement of the impurity of a node. Gini impurity can be measured by the following mathematical relationship:

$$G_i = 1 - \sum_{k=1}^{n} {p_{i,k}}^2$$

Figure 69: Gini-impurity equation, Geron (2017)

For example, if we consider the depth-2 left node would have a gini score of:

$$G_i = 1 - \{(0/54)^2 - (49/54)^2 - (5/54)^2\} \approx 0.168$$

Where,

$p_{i,k}$ = Ratio of class "*k*" instances among the training instances in the i_{th} node.

Similarly, the depth-1 left node has a "0" impurity or is completely "pure" because it applies only to Iris-setosa flower instances.

3.1.2.6.2.6 : Ensemble methods and Random Forests

Ensemble methods can be explained by an example as simple as asking a complex question to thousands of random people, then aggregating their answers. In many cases, we would see that the aggregated answers are better than an expert's answer. Similarly, an aggregated answer from a group of predictors

(classifiers or regressors), will often give us better predictions than with the best individual predictor. A group of predictors are called an "*Ensemble*", and this technique is therefore called, "*Ensemble learning*" and an Ensemble Learning algorithm is called "*Ensemble method"*.

In case of ML algorithms, we can take an example of a group of Decision Tree classifiers, each trained on a random subset of the training set. To get predictions from the classifiers, we need to obtain the predictions from each classifier on the part of training set it was trained on, and then predict the class that gets the most votes from all the Decision Tree classifiers combined. An ensemble of Decision Tree is called "*Random Forest*" (explained a bit more below) and is one of the most powerful ML algorithms.

With only a little difference, a Random Forest classifier has all the hyperparameters of a Decision Tree classifier that drives how trees are grown. Random Forest classifiers also introduces some extra randomness when growing trees i.e., in a Decision Tree the classifier searches for the very best feature when splitting a node, meanwhile, in Random Forest, the classifier searches for the best feature among a random subset of features.

3.1.2.6.2.7 : Voting Classifiers

Consider, that we trained a data set using some classifiers and we get an accuracy of let's 88% through the SVM classifier, 89% through the Logistic Regression, 91% from the Random Forest classifier and 80% from other classifiers.

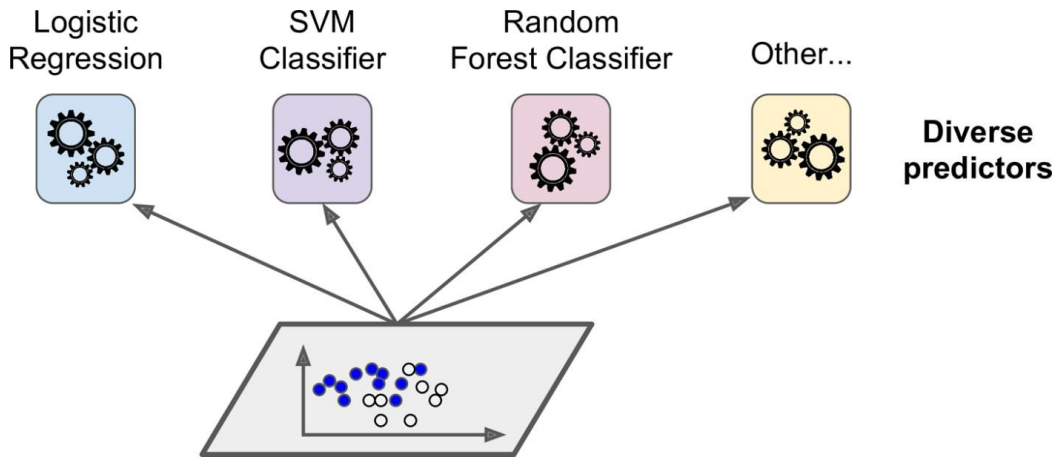

Figure 70: Working behind Voting Classifiers, Geron (2017)

In a Voting Classifier, we would aggregate the predictions of each of these classifiers and predict the class that gets the most votes. This would result in an even better classifier as the results would be based on majority-votes for each class. The majority-vote classifier is called a "*hard voting"* classifier and this as a result we might even get accuracy scores as good as 95% on this example of training set.

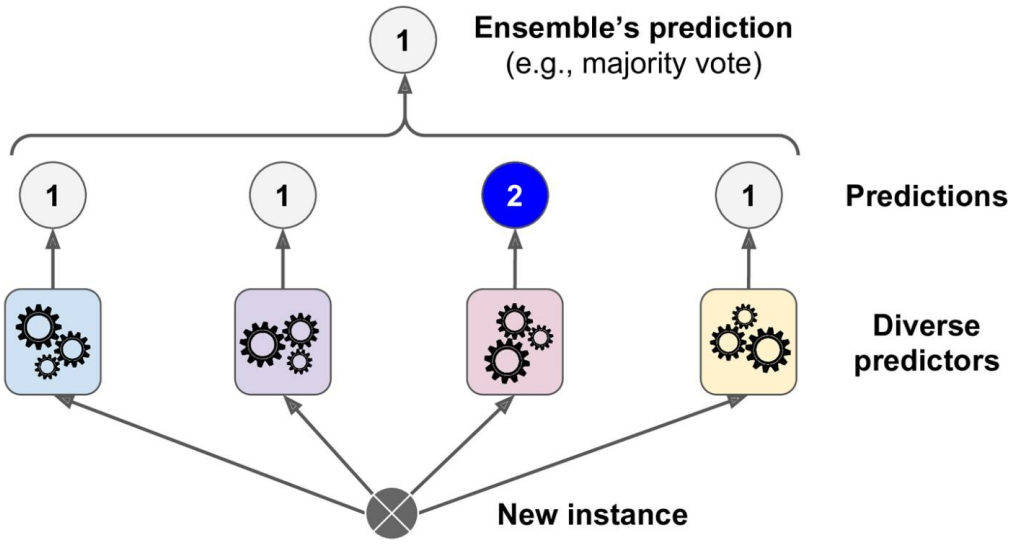

Figure 71: Working of an Ensemble method, Geron (2017)

Note that if all the classifiers are able to estimate class probabilities (also called confidence level of the classifier on each instance), then we are able to predict the class with the highest class probabilities, averaged over all the individual classifiers. This is called "soft voting" and this does even better than the "hard voting" as it gives more weight to the highest confident votes.

3.1.2.6.3 : *Accuracy measurements*

3.1.2.6.3.1 : Cross-validation

Cross-validation is a model validation technique for assessing how the model will generalize on the unseen instances of a new data. It is a useful tool for assessing how accurately a predictive model will do in practice. During training the predictive model, a model is given a dataset of known data usually and it is tested against a dataset that the model never saw during training (validation set or testing set). The purpose of cross-validation is to assign a dataset within the training dataset (called the validation set) that will be

used to test the model during training the model to verify that the model doesn't overfit during the training of the dataset and would eventually generalize on the test set.

3.1.2.6.3.1.1 : Cross-validation score

Scikit-learn provides its cross-validation feature among which K-fold cross validation is the most common one and used in our dataset. The number of folds "*K*" is represented by "*cv*" and we can calculate how well it does on each fold by calling Scikit-Learn's "*cross_val_score*". For example, if we had "*cv = 10*", the training set is split into 10 distinct subsets, then it trains and evaluates the selected model 10 times, picking a different fold for evaluation each time and training on the other 9 folds. By calling the cross_val_score we can then measure the accuracy on each fold of the training set.

3.1.2.6.3.1.2 : Cross-validation predictions

The cross-validation prediction has a similar working principle as *"cross_val_score*" but instead, for each time instead of giving scores on the predicted samples of each fold of training, it gives predictions. Scikit-Learn provides its "*cross_val_predict*" function to achieve this.

3.1.2.6.3.2 : Confusion matrix

It works on the idea of evaluating number of times instances of class 'X' are classified as class 'Y'. Once we call in the confusion matrix, we get a 2x2 matrix like this (generated by cross_val_predict on one of the classifier's training examples on our dataset):

```
array([[1034,   23],
       [  69,  264]], dtype=int64)
```

Figure 72: Confusion matrix outlook of a part of our data

Each row in a confusion matrix represents an actual class, while the columns represent the predicted class. The first row of the matrix considers the classes that were "*not shaly limestone*" (called the *negative class*), 1034 of the total instances were correctly classified as "*not shaly limestone*" (*true negatives*), while the remaining 23 instances were wrongly classified as "*shaly limestone*" (*false positives*).

The second row considers the "*shaly limestone*" class (*positive class*): 69 of the total were wrongly classified as "*not shaly limestone*" (*false negatives*), while the remaining 264 were correctly classified as "shaly limestone" (true positives). A perfect classifier would obviously have only true positives and true negatives only, and would have zero on the its main diagonals (top right to bottom left).

Here's a clearer picture of what confusion matrix is for an example of a classifier used for MNIST classification which involves classification of sample of handwritings from number 1-9.

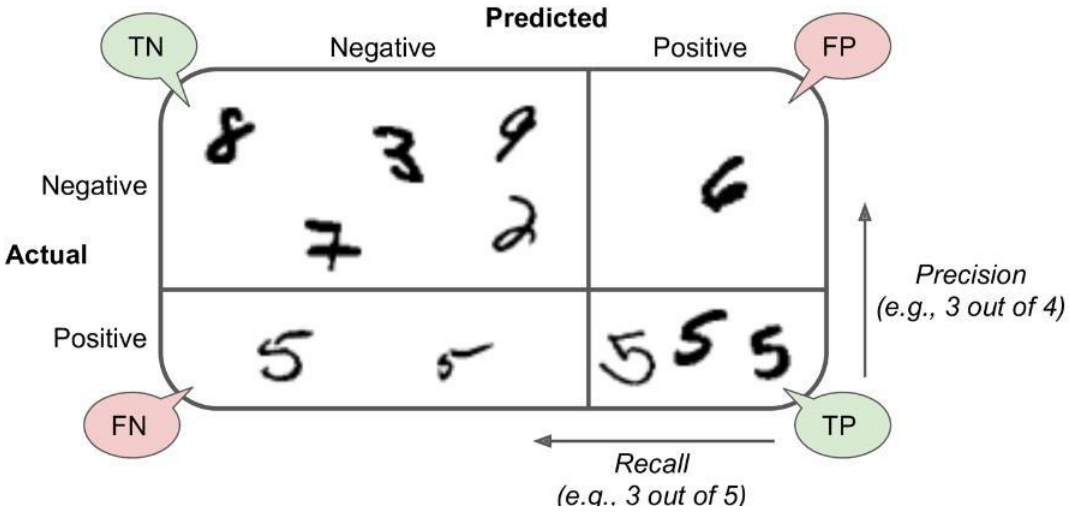

Figure 73: Explanation of confusion matrix, Geron (2017)

3.1.2.6.3.3 : Precision & Recall

The confusion matrix gives descriptive answers, which might a little too much when we need just information just as only the accuracy of positive predictions i.e. a more concise metric is required. This is what "*precision"* of the classifier does. It is given by:

$$\text{precision} = \frac{TP}{TP + FP}$$

Figure 74: Precision formula, Geron (2017)

As discussed earlier, TP is the number of true positive instances and FP is the false positive instances.

A perfect precision would mean that the classifier makes one single positive prediction and ensure it is correct (i.e. precision = 1/1 = 100%). This would not be useful information as the classifier would ignore all other instances except the positive instance. This brings "recall" into play, which is also known as "*sensitivity*" or "*True Positive Rate (TPR)*". Recall measures the ratio of positive instances that are correctly detected by the classifier.

$$\text{recall} = \frac{TP}{TP + FN}$$

Figure 75: Recall formula, Geron (2017)

Where, FN is the false negatives instances as discussed earlier.

The precision & recall scores have a specific tradeoff that depends on the set threshold. For example, in the MNIST classification, if we had a decision threshold set at different positions, here's what it will look like:

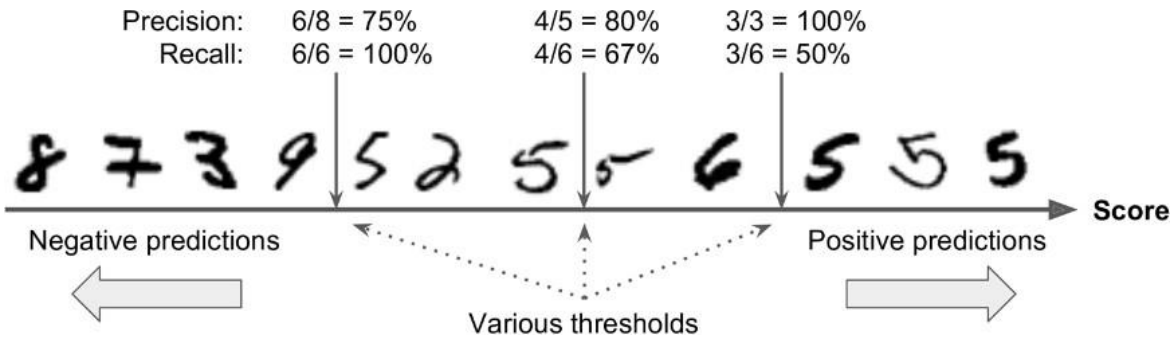

Figure 76: Setting a threshold and how does it work for a classification example, Geron (2017)

For each instance, the classifier computes a score dependent upon a decision function, if that score is greater than the set threshold, it classifies the instance to the positive or else it will go towards the negative class. As you can see that from the figure above, the precision increases from left to right, while the recall decreases. The explanation can be given my considering each threshold individually. If we had the threshold set at the center arrow, we would see that there are 4 true positives (positive predictions from center of the threshold arrow) and only one false positive (6 in this case). So with that threshold, we have 4/5 =80% precision. But on the other hand, we have 6 actual 5s in total and the classifier is only able to predict 4 of them correctly. Thus, recall is 4/6 = 67%.

Now, if the threshold is moved to the right arrow, we would have 3 out of 3 instances correctly classified as true positives, so precision would be 3/3 =100% ,while only 3 out of a total 6 were predict correctly so recall decreases further to 50% only. With this, depending on the project we would need to find the threshold that satisfies the minimum condition for the precision & recall. This is done by plotting the

precision and recall curves. With a simple function written for plotting the precision and recall with a threshold we can achieve this:

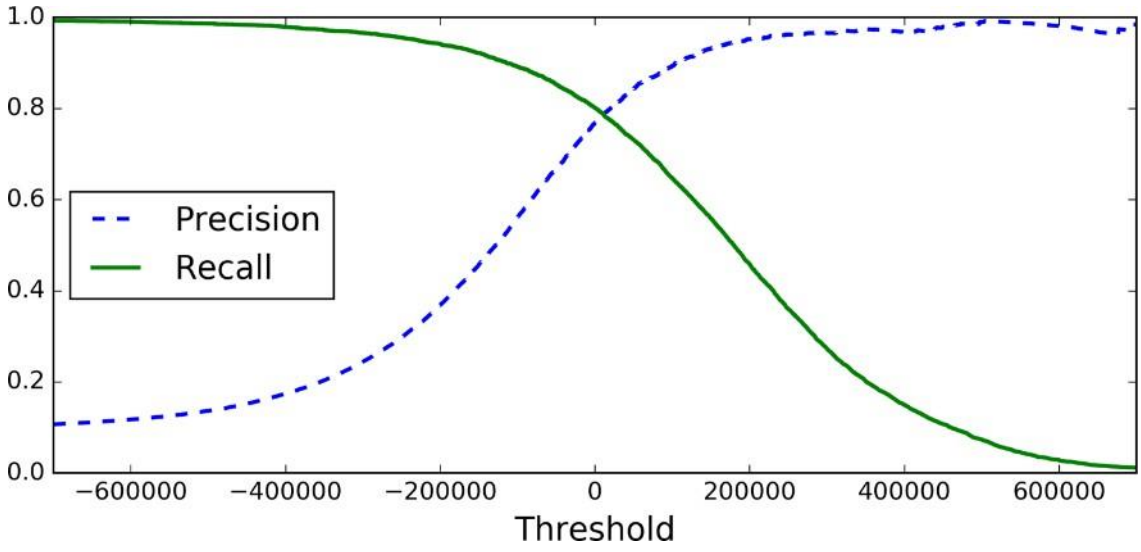

Figure 77: Precision-recall curve along with thresholds of a classification example, Geron (2017)

Note that the precision curve is bumpier towards the end. This is because that precision may sometimes go down when we sometimes increase the threshold as seen in the example above (when the threshold is moved to the left, precision goes down to 75% from 80%). While the recall curve is smooth throughout since recall only goes down when the threshold is increased.

With this, we can simply select a threshold value that gives the best precision/recall for our task. Also, another way to select the threshold value would be to plot precision again recall directly:

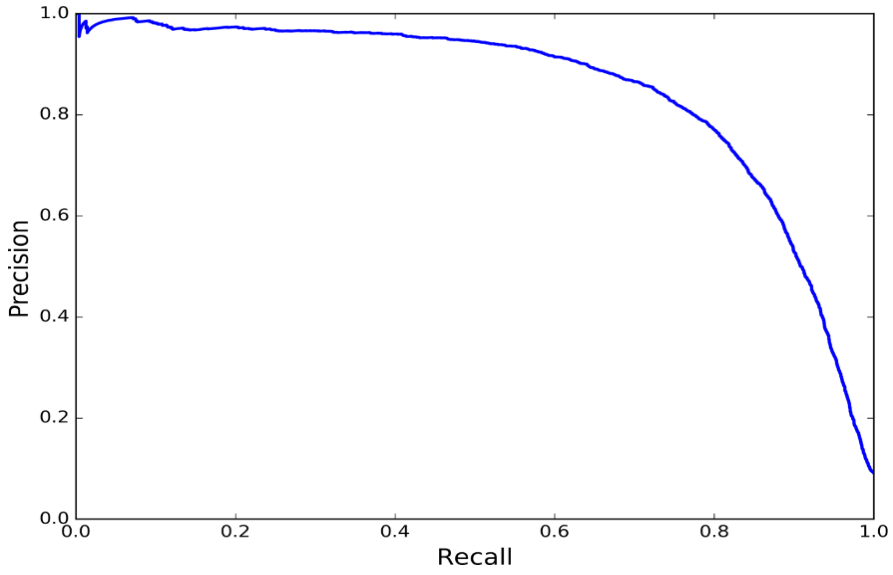

Figure 78: Precision-recall curve for a classification example

As can be seen, recall starts to drop abruptly from 80% recall and we might want to select a precision/recall tradeoff just before that drop, somewhere around 60% but again it depends upon the nature of the project we are dealing with.

3.1.2.6.3.4 : F1-score

A convenient way to combine precision and recall and take both into account while converting it into a single metric is called F1-score to compare two classifiers. To F1-score is basically is a harmonic mean of precision and recall which is better than the simple mean which gives equal weightage to all values, the harmonic mean gives more weight to low values. Consequently, the classifier would only give a high F1-score only if both precision and recall are high. Therefore, you would see this accuracy score being used repeatedly in our code to test the predictions from the classifiers.

$$F_1 = \frac{2}{\frac{1}{\text{precision}} + \frac{1}{\text{recall}}} = 2 \times \frac{\text{precision} \times \text{recall}}{\text{precision} + \text{recall}} = \frac{TP}{TP + \frac{FN + FP}{2}}$$

Figure 79: Formula for F1 scores, Geron (2017)

You will often find in our code the term “average = weighted” being used again and again in our F1-scores, the reason simply being that weighted calculates the metrics for each label, finds their average, weighted by the number of true instances for each label and also accounts for label imbalance.

3.1.2.6.3.5 : ROC curves and ROC-AUC scores

Receiver Operating Characteristics (ROC) curve is another important and commonly used accuracy measurement tool specially used for binary classifiers. It is similar to to the precision/recall curve, but instead of plotting precision vs recall, the ROC curves plot the TPR and FPR. To plot the ROC curves we would need to first obtain the FPR & TPR which is what you would see in our codes repeatedly before plotting the ROC curves. This measurement tool is used most commonly in our codes due to its accountancy of many of the other accuracy measurements we have discussed so far.

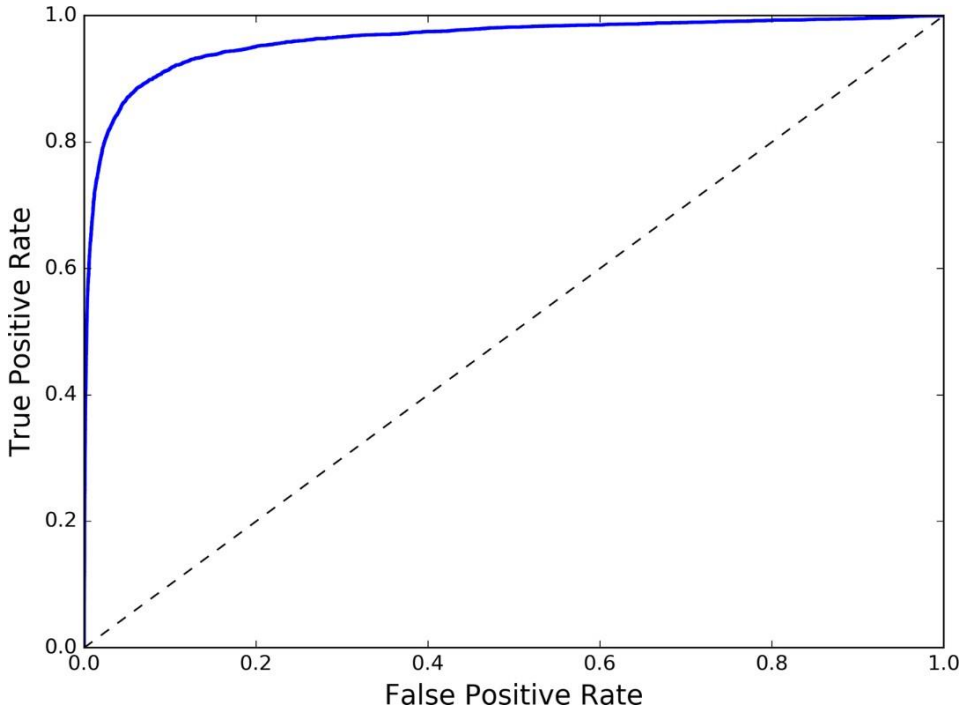

Figure 80: ROC curve for a single classifier on a classification example, Geron (2017)

ROC-AUC score is another alternative way to evaluate the classifiers. A perfect classifier would always have a ROC AUC equal to 1 while, a random classifier with an imperfect classification would have an ROC-AUC of just 0.5.

3.1.2.6.4 : *Grid Search*

To improve the performance of the classifiers, one good way would be to play with the hyper-parameters of the classifiers until we find the right combination that gives the best accuracy scores. But this would be a lot of work as there are several combinations of these hyper-parameters possible and this would be

tedious work as we might not have enough time during the project to explore all these possible combinations.

With Grid Search, this is very easy to implement. This is done by simply giving Grid Search a possible combination of hyper-parameter values we want to experiment with for the classifier and using cross-validation (discussed earlier) it would evaluate each combination outputs the best parameters based on the cross-validation scores.

3.1.2.6.5 *: Training the dataset*

3.1.2.6.5.1 : Training shaly limestone class

Since most of the background of the ML algorithms have been discussed already the code implementation along with some more explanation about the codes will be discussed in this section.

Since we already had the data converted to Boolean form, we decided to train each class individually as shown earlier as well. The codes shown below are used to train the "shaly limestone" class as "shaly limestone" or "not shaly limestone".

Take note that, each time to get the predictions from the classifiers, all we need to do is fit the data with the classifiers (i.e., by .fit) and then predict (by .predict). You will see this repeated over and over in our codes. Let's import some of the classifiers we discussed earlier and check their performances using Scikit-Learn's libraries:

LogisticRegression on training set

```
In [84]: from sklearn.linear_model import LogisticRegressionCV

         log_reg_n = LogisticRegressionCV(random_state=42)

In [85]: cross_val_score(log_reg_n, X_train, y_train_sl, cv=5, scoring ="f1_weighted")

Out[85]: array([ 0.72389625,  0.75531066,  0.76169739,  0.73720477,  0.77486259])

In [86]: y_predLog = log_reg_n.fit(X_train, y_train_sl).predict(X_train)
         ("Number of mislabeled points out of a total %d points : %d" % (X_train.shape[0],(y_train_sl != y_predLog).sum()))

Out[86]: 'Number of mislabeled points out of a total 1390 points : 299'

In [87]: from sklearn.model_selection import cross_val_predict
         y_log_crpred =cross_val_predict(log_reg_n, X_train, y_train_sl, cv=5, method ="predict")

In [282]: from sklearn.metrics import f1_score

          f1_score(y_train_sl, y_log_crpred, average = "weighted")

Out[282]: 0.75125598355975243
```

Figure 81: Training Logistic regression on data

```
In [118]: y_train_slpred3 = cross_val_predict(log_reg_n, X_train, y_train_sl, cv=5)

In [119]: confusion_matrix(y_train_sl, y_train_slpred3,)

Out[119]: array([[985,  72],
                 [235,  98]], dtype=int64)

In [120]: precision_score(y_train_sl, y_train_slpred3,average = 'weighted')

Out[120]: 0.75205917040716497

In [121]: recall_score(y_train_sl, y_train_slpred3,average = 'weighted')

Out[121]: 0.77913669064748203
```

Figure 82: Scores with Logistic regression

```
In [138]: y_logscores = cross_val_predict(log_reg_n, X_train, y_train_sl, cv=5,
                                          method="decision_function")

In [139]: y_logscores

Out[139]: array([[ 0.        , -1.94908639],
                 [ 0.        , -5.98252164],
                 [ 0.        , -1.88489191],
                 ...,
                 [ 0.        , -4.16359345],
                 [ 0.        , -4.05007574],
                 [ 0.        , -1.5577092 ]])
```

Figure 83: ROC-curve preparation for Logistic regression

```
In [141]: #hack to work around issue #9589 introduced in Scikit-Learn 0.19.0
          if y_logscores.ndim == 2:
              y_logscores = y_logscores[:, 1]

In [142]: from sklearn.metrics import precision_recall_curve

          precisions, recalls, thresholds = precision_recall_curve(y_train_sl, y_logscores)
```

Figure 84: Precision-recall curve parameters with Logistic regression

Notice here we have used index slicing of the log scores for decision function from cross validation prediction to call the precision, recall curve for right plotting of the curve with reference to working a way around the classifiers which has decision function instead.

Using these values, we can make our function of plotting the precision and recalls:

In [144]:
```
def plot_precision_recall_vs_threshold(precisions, recalls, thresholds):
    plt.plot(thresholds, precisions[:-1], "b--", label="Precision", linewidth=2)
    plt.plot(thresholds, recalls[:-1], "g-", label="Recall", linewidth=2)
    plt.xlabel("Threshold", fontsize=16)
    plt.legend(loc="lower right", fontsize=16)
    plt.ylim([0, 1.1])

plt.figure(figsize=(8, 4))
plot_precision_recall_vs_threshold(precisions, recalls, thresholds)
plt.xlim([-5,5])
plt.show()
```

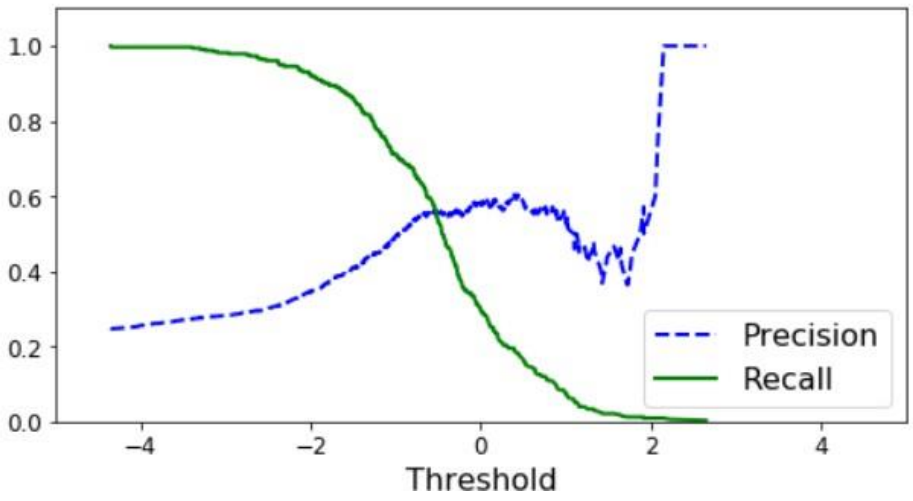

Figure 85: Precision-recall curve with thresholds of Logistic regression on data

Let's plot the precision against recall for even better visualization for selection of a threshold:

```
In [145]: def plot_precision_vs_recall(precisions, recalls):
              plt.plot(recalls, precisions, "b-", linewidth=2)
              plt.xlabel("Recall", fontsize=16)
              plt.ylabel("Precision", fontsize=16)
              plt.axis([0, 1, 0, 1])

          plt.figure(figsize=(8, 6))
          plot_precision_vs_recall(precisions, recalls)
          plt.show()
```

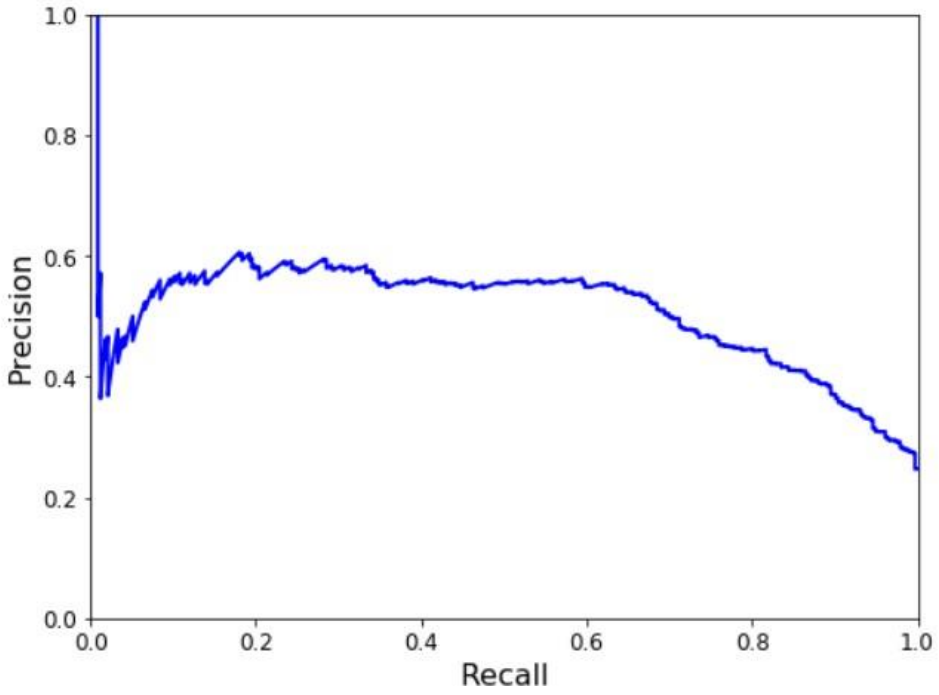

Figure 86: Precision-recall curve of Logistic regression on dataset

As seen from the plot, precision falls sharply at very low recall, thus the classifier seems to do very bad on the training set and setting a threshold would a difficult choice in this case. Instead of plotting precision, recall curves for all the classifiers as this would require selecting a threshold value each time we decided to select a better accuracy measurement technique which also accounts for both precision and recalls without needing to calculate a threshold each time based on precision recall curves i.e., the roc-curves (discussed earlier).

```
In [146]: from sklearn.metrics import roc_auc_score
          roc_auc_score(y_train_sl, y_logscores, average = 'weighted')

Out[146]: 0.80596395828183887
```

Figure 87: ROC-AUC scores of Logistic Regression on data

```
In [153]: fpr_log, tpr_log, thresholds_log = roc_curve(y_train_sl, y_logscores)
```

Figure 88: False-Positives, True-positive and thresholds for ROC-curve of Logistic regression on training

Similarly,

Gaussian Naive Bayes on training set:

```
In [103]: from sklearn.naive_bayes import GaussianNB
          gnb_clf_n = GaussianNB()
          y_predNB = gnb_clf_n.fit(X_train, y_train_sl)

In [283]: cross_val_score(gnb_clf_n, X_train, y_train_sl, cv=5, scoring ="f1_weighted")

Out[283]: array([ 0.73777109,  0.74831147,  0.69594624,  0.76289201,  0.70333631])

In [105]: y_predNB_tr = gnb_clf_n.fit(X_train, y_train_sl).predict(X_train)
          ("Number of mislabeled points out of a total %d points : %d" % (X_train.shape[0],(y_train_sl != y_predNB_tr).sum()))

Out[105]: 'Number of mislabeled points out of a total 1390 points : 395'

In [106]: y_predgnb_cv = cross_val_predict(gnb_clf_n, X_train, y_train_sl, cv=5, method = "predict")

In [107]: f1_score(y_train_sl, y_predgnb_cv, average = 'weighted')

Out[107]: 0.73014854303607479
```

Figure 89: Training Gaussian Naive Bayes on data

Gaussian Naive Bayes:

```
In [133]: y_train_slpred6 = cross_val_predict(gnb_clf_n, X_train, y_train_sl, cv=5)

In [134]: confusion_matrix(y_train_sl, y_train_slpred6)

Out[134]: array([[774, 283],
                 [115, 218]], dtype=int64)

In [135]: precision_score(y_train_sl, y_train_slpred6, average = 'weighted')

Out[135]: 0.76630641735845129

In [136]: recall_score(y_train_sl, y_train_slpred6,average = 'weighted')

Out[136]: 0.71366906474820146

In [137]: f1_score(y_train_sl, y_train_slpred6, average = 'weighted')

Out[137]: 0.73014854303607479
```

Figure 90: Scores with Gaussian Naive Bayes on training data

```
In [154]: y_probas_gnb = cross_val_predict(gnb_clf_n, X_train, y_train_sl, cv =5, method = "predict_proba")

In [155]: y_scores_gnb = y_probas_gnb[:,1]
          fpr_gnb, tpr_gnb, thresholds_gnb = roc_curve(y_train_sl, y_scores_gnb)

In [156]: roc_auc_score(y_train_sl, y_scores_gnb, average = 'weighted')

Out[156]: 0.73798869825359892
```

Figure 91: Scores with Gaussian Naive Bayes on training data

GaussianRBF SVM Classsifier on training set:

```
In [95]: from sklearn.svm import SVC

         rbf_kernel_svm_clf = SVC(C=1, probability = True)
         rbf_kernel_svm_clf.fit(X_train, y_train_sl)

Out[95]: SVC(C=1, cache_size=200, class_weight=None, coef0=0.0,
           decision_function_shape='ovr', degree=3, gamma='auto', kernel='rbf',
           max_iter=-1, probability=True, random_state=None, shrinking=True,
           tol=0.001, verbose=False)

In [285]: cross_val_score(rbf_kernel_svm_clf, X_train, y_train_sl, cv=5, scoring ="f1_weighted")

Out[285]: array([ 0.89339565,  0.86639757,  0.88002552,  0.85966672,  0.89526028])

In [97]: y_predsvm = rbf_kernel_svm_clf.fit(X_train, y_train_sl).predict(X_train)
         ("Number of mislabeled points out of a total %d points : %d" % (X_train.shape[0],(y_train_sl != y_predsvm).sum()))

Out[97]: 'Number of mislabeled points out of a total 1390 points : 127'

In [98]: y_svm_cr = cross_val_predict(rbf_kernel_svm_clf, X_train, y_train_sl, cv =5, method = "predict")

In [99]: f1_score(y_train_sl, y_svm_cr, average = 'weighted')

Out[99]: 0.87894265229034829
```

Figure 92: Training SVM Classifier on training data

```
In [128]: y_train_slpred5 = cross_val_predict(rbf_kernel_svm_clf, X_train, y_train_sl, cv=5)

In [129]: confusion_matrix(y_train_sl, y_train_slpred5)

Out[129]: array([[1003,   54],
                 [ 109,  224]], dtype=int64)

In [130]: precision_score(y_train_sl, y_train_slpred5, average = 'weighted')

Out[130]: 0.87892642720356096

In [131]: recall_score(y_train_sl, y_train_slpred5, average = 'weighted')

Out[131]: 0.88273381294964026

In [132]: f1_score(y_train_sl, y_train_slpred5, average = 'weighted')

Out[132]: 0.87894265229034829
```

Figure 93: Scores of training Gaussian SVM classifier on training data

```
In [160]: y_svmscores = cross_val_predict(rbf_kernel_svm_clf, X_train, y_train_sl, cv =5, method = "decision_function")

In [161]: #hack to work around issue #9589 introduced in Scikit-Learn 0.19.0
          if y_svmscores.ndim == 2:
              y_svmscores = y_svmscores[:, 1]

In [162]: fpr_svm, tpr_svm, thresholds_svm = roc_curve(y_train_sl, y_svmscores)

In [163]: roc_auc_score(y_train_sl, y_svmscores, average = 'weighted')

Out[163]: 0.92393339413206976
```

Figure 94: Scores of training Gaussian SVM classifier on training data

KNeighborsClassifier on training set:

```
In [100]: from sklearn.neighbors import KNeighborsClassifier
          knn_clf_n = KNeighborsClassifier()
          knn_clf_n.fit(X_train, y_train_sl)

Out[100]: KNeighborsClassifier(algorithm='auto', leaf_size=30, metric='minkowski',
                     metric_params=None, n_jobs=1, n_neighbors=5, p=2,
                     weights='uniform')

In [286]: cross_val_score(knn_clf_n, X_train, y_train_sl, cv=5, scoring ="f1_weighted")

Out[286]: array([ 0.91093171,  0.88642074,  0.89976849,  0.88987724,  0.85806555])

In [102]: y_predknn = knn_clf_n.fit(X_train, y_train_sl).predict(X_train)
          ("Number of mislabeled points out of a total %d points : %d" % (X_train.shape[0],(y_train_sl != y_predknn).sum()))

Out[102]: 'Number of mislabeled points out of a total 1390 points : 85'
```

Figure 95: Training K-Nearest Neighbors on dataset

K-Nearest Neighbor Classifier:

```
In [123]: y_train_slpred4 = cross_val_predict(knn_clf_n, X_train, y_train_sl, cv=5)

In [124]: confusion_matrix(y_train_sl, y_train_slpred4)

Out[124]: array([[995,  62],
                 [ 90, 243]], dtype=int64)

In [125]: precision_score(y_train_sl, y_train_slpred4, average = 'weighted')

Out[125]: 0.88822358083918906

In [126]: recall_score(y_train_sl, y_train_slpred4,average = 'weighted')

Out[126]: 0.89064748201438848

In [127]: f1_score(y_train_sl, y_train_slpred4, average = 'weighted')

Out[127]: 0.88896261931999676
```

Figure 96: Training scores with K-Nearest Neighbors classifier on data

```
In [157]: y_probas_knn = cross_val_predict(knn_clf_n, X_train, y_train_sl, cv =5, method = "predict_proba")

In [158]: y_scores_knn = y_probas_knn[:,1]
          fpr_knn, tpr_knn, thresholds_knn = roc_curve(y_train_sl, y_scores_knn)

In [159]: roc_auc_score(y_train_sl, y_scores_knn, average = 'weighted')

Out[159]: 0.91793449078217293
```

Figure 97: Training scores with K-Nearest Neighbors classifier on data

DecisionTree on training set

```
In [92]: from sklearn.tree import DecisionTreeClassifier
         tree_clf_n = DecisionTreeClassifier(random_state =42)
         tree_clf_n.fit(X_train, y_train_sl)

Out[92]: DecisionTreeClassifier(class_weight=None, criterion='gini', max_depth=None,
                     max_features=None, max_leaf_nodes=None,
                     min_impurity_decrease=0.0, min_impurity_split=None,
                     min_samples_leaf=1, min_samples_split=2,
                     min_weight_fraction_leaf=0.0, presort=False, random_state=42,
                     splitter='best')

In [287]: cross_val_score(tree_clf_n, X_train, y_train_sl, cv=5, scoring ="f1_weighted")

Out[287]: array([ 0.94321037,  0.91592645,  0.93032874,  0.91581915,  0.89419589])

In [94]: y_predtree = tree_clf_n.fit(X_train, y_train_sl).predict(X_train)
         ("Number of mislabeled points out of a total %d points : %d" % (X_train.shape[0],(y_train_sl != y_predtree).sum()))

Out[94]: 'Number of mislabeled points out of a total 1390 points : 0'
```

Figure 98: Training Decision Tree classifier on dataset

```
In [113]: y_train_slpred2 = cross_val_predict(tree_clf_n, X_train, y_train_sl, cv=5)

In [114]: confusion_matrix(y_train_sl, y_train_slpred2)

Out[114]: array([[995,  62],
                 [ 50, 283]], dtype=int64)

In [115]: precision_score(y_train_sl, y_train_slpred2,average = 'weighted')

Out[115]: 0.92056284824439183

In [116]: recall_score(y_train_sl, y_train_slpred2,average = 'weighted')

Out[116]: 0.91942446043165471

In [117]:
          f1_score(y_train_sl, y_train_slpred2, average = 'weighted')

Out[117]: 0.91990752152115784
```

Figure 99: Training scores with Decision Tree Classifier on data

```
y_probas_tree = cross_val_predict(roc_tree, X_train, y_train_sl, cv =5, method = "predict_proba")

In [151]: y_scores_tree = y_probas_tree[:,1] #score = proba of positive class
          fpr_tree, tpr_tree, thresholds_tree = roc_curve(y_train_sl, y_scores_tree)

In [152]: roc_auc_score(y_train_sl, y_scores_tree, average = 'weighted')

Out[152]: 0.8955966373184916
```

Figure 100: Training scores with Decision Tree Classifier on data

RandomForest on training set

In [89]:
```
from sklearn.ensemble import RandomForestClassifier

forest_clf_n = RandomForestClassifier(random_state =42)
forest_clf_n.fit(X_train, y_train_sl)
```

Out[89]:
```
RandomForestClassifier(bootstrap=True, class_weight=None, criterion='gini',
            max_depth=None, max_features='auto', max_leaf_nodes=None,
            min_impurity_decrease=0.0, min_impurity_split=None,
            min_samples_leaf=1, min_samples_split=2,
            min_weight_fraction_leaf=0.0, n_estimators=10, n_jobs=1,
            oob_score=False, random_state=42, verbose=0, warm_start=False)
```

In [288]:
```
cross_val_score(forest_clf_n, X_train, y_train_sl, cv=5, scoring ="f1_weighted")
```

Out[288]:
```
array([ 0.95980709,  0.92280374,  0.92204216,  0.92467386,  0.9304593 ])
```

In [91]:
```
y_predfor_p_n = forest_clf_n.fit(X_train, y_train_sl).predict(X_train)
("Number of mislabeled points out of a total %d points : %d" % (X_train.shape[0],(y_train_sl != y_predfor_p_n).sum()))
```

Out[91]:
```
'Number of mislabeled points out of a total 1390 points : 4'
```

Figure 101: Training Random Forest Classifier on dataset

```
In [108]: from sklearn.model_selection import cross_val_predict

          y_train_slpred = cross_val_predict(forest_clf_n, X_train, y_train_sl, cv=5)

In [109]: from sklearn.metrics import confusion_matrix

          confusion_matrix(y_train_sl, y_train_slpred)

Out[109]: array([[1034,   23],
                 [  69,  264]], dtype=int64)

In [110]: from sklearn.metrics import precision_score, recall_score

          precision_score(y_train_sl, y_train_slpred, average = 'weighted')

Out[110]: 0.93323107581138565

In [111]: recall_score(y_train_sl, y_train_slpred,average = 'weighted')

Out[111]: 0.93381294964028771

In [112]: from sklearn.metrics import f1_score

          f1_score(y_train_sl, y_train_slpred, average = 'weighted')

Out[112]: 0.93206239309627548
```

Figure 102: Training scores with Random Forest Classifier on data

```
In [147]: from sklearn.ensemble import RandomForestClassifier

          roc_forest = RandomForestClassifier(random_state = 42)
          y_probas_forest = cross_val_predict(roc_forest, X_train, y_train_sl, cv =5, method = "predict_proba")

In [148]: from sklearn.metrics import roc_curve

          y_scores_forest = y_probas_forest[:,1] #score = proba of positive class
          fpr_forest, tpr_forest, thresholds_forest= roc_curve(y_train_sl, y_scores_forest)

In [150]: roc_auc_score(y_train_sl, y_scores_forest, average ='weighted') #Best ROC-AUC Score for forest

Out[150]: 0.97054670564604339
```

Figure 103: Training scores with Random Forest classifier on data

As we can see from f1 & roc curves that we get almost perfect score on Random Forest, SVM and K-Nearest Neighbor classifiers. We can now make our modified function (Geron, 2017) to plot the ROC-Curves and verify the performance of these classifiers.

```
In [164]: def plot_roc_curve(fpr, tpr, label=None):
              plt.plot(fpr, tpr, linewidth=2, label=label)
              plt.plot([0, 1], [0, 1], 'k--')
              plt.axis([0, 1, 0, 1])
              plt.xlabel('False Positive Rate', fontsize=16)
              plt.ylabel('True Positive Rate', fontsize=16)
```

Figure 104: Function for ROC-curve plotting, Geron (2017)

```
In [165]: plt.figure(figsize=(8, 6))
plt.plot(fpr_log, tpr_log, "b:", linewidth=2, label="Logistic Regressor")
plt.plot(fpr_svm, tpr_svm, "m.", linewidth=1, label="SVM Classifier")
plot_roc_curve(fpr_forest, tpr_forest, "Random Forest")
plot_roc_curve(fpr_tree, tpr_tree, "Decision Tree")
plot_roc_curve(fpr_gnb, tpr_gnb, "Naive Bayes")
plot_roc_curve(fpr_knn, tpr_knn, "K-Nearest Neighbor")
plt.legend(loc="lower right", fontsize=16)
plt.show()
```

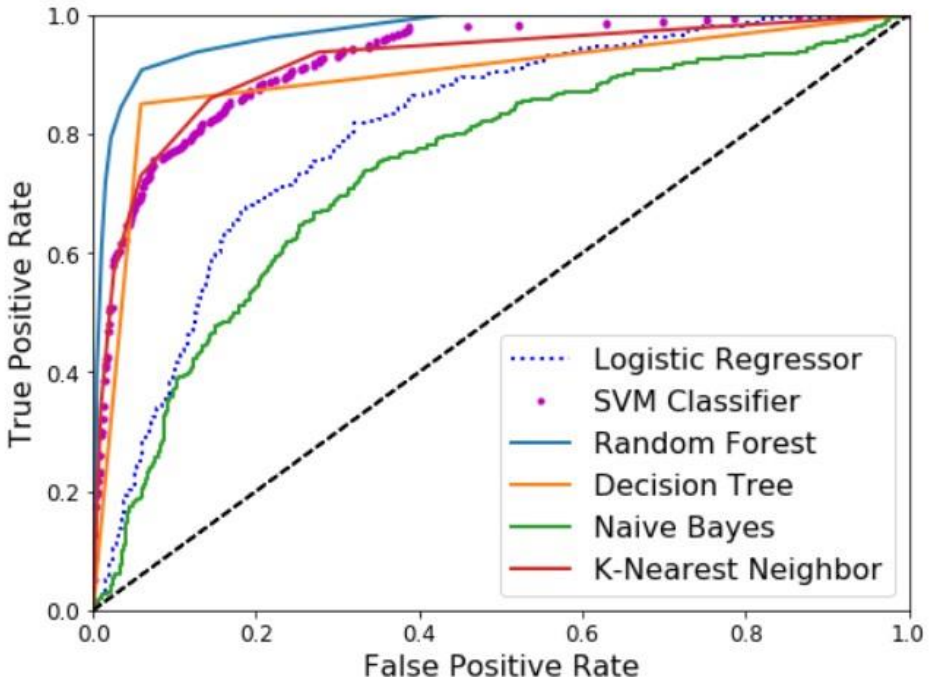

Figure 105: ROC-curves of all the classifier on Shaly-Limestone class for training data

This confirms that the three best classifiers were indeed Random Forest, K-Nearest Neighbor and SVM indeed. Before we proceed, we did Grid Search on hyper-parameters of these classifiers to improve their performance even more and verify at the end on the test set. The purpose of keeping all three of these classifiers instead of just Random Forest classifier alone is to later combine these classifiers together in Ensemble methods & Voting classifier to further improve the performance on the test set after Grid Search on their individual hyper-parameters.

```
In [165]: plt.figure(figsize=(8, 6))
          plt.plot(fpr_log, tpr_log, "b:", linewidth=2, label="Logistic Regressor")
          plt.plot(fpr_svm, tpr_svm, "m.", linewidth=1, label="SVM Classifier")
          plot_roc_curve(fpr_forest, tpr_forest, "Random Forest")
          plot_roc_curve(fpr_tree, tpr_tree, "Decision Tree")
          plot_roc_curve(fpr_gnb, tpr_gnb, "Naive Bayes")
          plot_roc_curve(fpr_knn, tpr_knn, "K-Nearest Neighbor")
          plt.legend(loc="lower right", fontsize=16)
          plt.show()
```

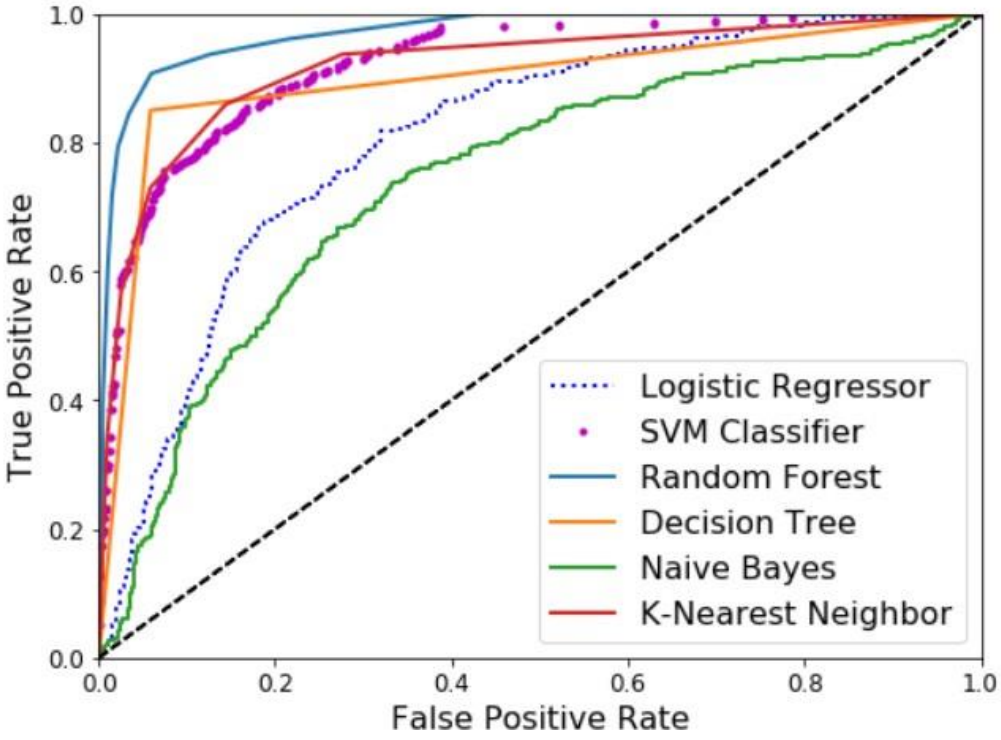

Figure 106: ROC curves for all the classifier on test set

As you can that the Logistic Regression does not do very well on the training set, which explains a weird looking precision recall curve. On the other hand, Naïve Bayes classifier does not do a good job either, but, the Random Forest classifier, Decision Tree classifier, K-Nearest Neighbor classifier, and SVM classifier (in order of decreasing performance on the shaly limestone training set), does very well on the training set. With that we decided to improve the three best classifiers here, namely, Random Forest, SVM and K-Nearest Neighbors (Decision Tree was dropped as Random Forest after all is an ensemble of Decision Tree classifier).

3.1.2.6.5.2 : Grid Search on best classifiers

As already discussed, Grid Search involves selecting the best hyperparameters that gives the best cross-validation results among a several combinations of the hyperparameters. Each classifier has its own hyperparameter. For example, in a K-Nearest Neighbor classifier as discussed already, the "*K-neighbors*" and "*weights*" are such parameters which we cannot be sure about what values would yield the best cross-validation scores. Thus, we would want to implement Grid Search on this as follows:

```
In [166]: from sklearn.model_selection import GridSearchCV

          weights_sl = ['uniform','distance']
          numNeighbors_sl = np.array([3,5,7,9])

In [167]: param_grid_knn_sl  = dict(weights=weights_sl,n_neighbors=numNeighbors_sl)

In [168]: grid_knn_sl = GridSearchCV(knn_clf_n,param_grid=param_grid_knn_sl,cv=5, n_jobs =-1)

In [169]: grid_knn_sl.fit(X_train,y_train_sl)

Out[169]: GridSearchCV(cv=5, error_score='raise',
                estimator=KNeighborsClassifier(algorithm='auto', leaf_size=30, metric='minkowski',
                    metric_params=None, n_jobs=1, n_neighbors=5, p=2,
                    weights='uniform'),
                fit_params=None, iid=True, n_jobs=-1,
                param_grid={'weights': ['uniform', 'distance'], 'n_neighbors': array([3, 5, 7, 9])},
                pre_dispatch='2*n_jobs', refit=True, return_train_score=True,
                scoring=None, verbose=0)
```

Figure 107: Grid Search on K-Nearest Neighbors classifier on Shaly-Limestone data

Note that the hyperparameter of Grid Search "*n_jobs =-1*" would use all the cores of CPU to make the process fast as Grid Search when implemented on a number of hyperparameters takes a lot of time.

Now, once we fit our data on Grid Search, we get the following:

```
In [170]: grid_knn_sl.best_params_
Out[170]: {'n_neighbors': 5, 'weights': 'distance'}

In [171]: grid_knn_sl.best_estimator_
Out[171]: KNeighborsClassifier(algorithm='auto', leaf_size=30, metric='minkowski',
                               metric_params=None, n_jobs=1, n_neighbors=5, p=2,
                               weights='distance')
```

Figure 108: Best parameters of K-Nearest Neighbors Classifier

This tells us that if we use the "*K-neighbors* or *n_neighbors*" value of 5 and with "*weights*" set as "distance" we would have the best cross-validation scores. Also, if we want to directly use the classifier setup with Grid Search directly without needing to input the hyperparameters manually, we can simply use the Grid Search's "*best_estimator_*".

Now we can verify if Grid Search was actually really useful or not.

Comparing without grid search:

```
In [178]: y_test_knn_noGS = knn_clf_n.predict(X_test_prepared)

In [179]: f1_score(y_test_sl, y_test_knn_noGS, average = 'weighted')
Out[179]: 0.90498621431611748

In [180]: roc_auc_score(y_test_sl, y_test_knn_noGS, average = 'weighted')
Out[180]: 0.87055126928820303
```

Figure 109: Scores without Grid Search parameters of K-Nearest Neighbors

With Grid Search:

```
In [172]: knn_clf_sl_GS = KNeighborsClassifier(n_neighbors = 5, weights = 'distance')

In [173]: knn_clf_sl_GS.fit(X_train, y_train_sl)

Out[173]: KNeighborsClassifier(algorithm='auto', leaf_size=30, metric='minkowski',
                               metric_params=None, n_jobs=1, n_neighbors=5, p=2,
                               weights='distance')

In [174]: y_pred_knn_sl  = cross_val_predict(knn_clf_sl_GS, X_train, y_train_sl, cv=5)
          f1_score(y_train_sl, y_pred_knn_sl , average="weighted")

Out[174]: 0.90076522893834454

In [175]: y_test_knn_pred = knn_clf_sl_GS.predict(X_test_prepared)

In [176]: f1_score(y_test_sl, y_test_knn_pred , average="weighted")

Out[176]: 0.91092555406127373

In [177]: roc_auc_score(y_test_sl, y_test_knn_pred, average = 'weighted')

Out[177]: 0.87995686079309776
```

Figure 110: Scores with Grid Search parameters of K-Nearest Neighbors classifier

Note that we used our test sets to evaluate the classifiers. This is because since we have already now tested the classifiers on cross validation prediction and run Grid Search on the classifiers, it is now safe to use the test sets. Also, it is clearly visible that with Grid Search suggested hyperparameters we had an increase F1 & ROC-AUC score.

Similarly, we can do this for the other two classifiers as well.

```
In [184]: Cs = [5, 100, 200]
          kernels = ['linear', 'rbf']
          decision = ['ovo', 'ovr']

In [185]: param_grid_svm_sl  = dict(C=Cs,kernel = kernels, decision_function_shape =decision)

In [186]: grid_svm_sl = GridSearchCV(rbf_kernel_svm_clf,param_grid=param_grid_svm_sl,cv=5, n_jobs =-1)

In [187]: grid_svm_sl.fit(X_train, y_train_sl)

Out[187]: GridSearchCV(cv=5, error_score='raise',
                estimator=SVC(C=1, cache_size=200, class_weight=None, coef0=0.0,
              decision_function_shape='ovr', degree=3, gamma='auto', kernel='rbf',
              max_iter=-1, probability=True, random_state=None, shrinking=True,
              tol=0.001, verbose=False),
                fit_params=None, iid=True, n_jobs=-1,
                param_grid={'C': [5, 100, 200], 'kernel': ['linear', 'rbf'], 'decision_function_shape': ['ovo', 'ovr']},
                pre_dispatch='2*n_jobs', refit=True, return_train_score=True,
                scoring=None, verbose=0)

In [188]: grid_svm_sl.best_params_

Out[188]: {'C': 100, 'decision_function_shape': 'ovo', 'kernel': 'rbf'}

In [189]: grid_svm_sl.best_estimator_

Out[189]: SVC(C=100, cache_size=200, class_weight=None, coef0=0.0,
            decision_function_shape='ovo', degree=3, gamma='auto', kernel='rbf',
            max_iter=-1, probability=True, random_state=None, shrinking=True,
            tol=0.001, verbose=False)
```

Figure 111: Grid Search on SVM and best parameters on Shaly Limestone data

Note that in the SVM classifier, we did a Grid Search on the hyperparameter "*decision function*" with parameter values as *"OneVsRest (OvR)"* and *"OneVsOne (OvO)"*. OvR basically involves training a single classifier per class while the OvO deals involves training a separate classifier for each different pair of labels. In simple words, if we were dealing with a case where we were classifying all the formations together OvR would have been the better choice. But since, right now we are classifying only a shaly limestone class with labels as 0's and 1's (Boolean form), so it would be best if we use an OvO kernel, which is what the Grid Search itself tells us.

```
In [190]: svm_clf_sl_GS = SVC(C= 100, decision_function_shape = 'ovo', random_state =41, probability = True)

In [191]: svm_clf_sl_GS.fit(X_train, y_train_sl)

Out[191]: SVC(C=100, cache_size=200, class_weight=None, coef0=0.0,
            decision_function_shape='ovo', degree=3, gamma='auto', kernel='rbf',
            max_iter=-1, probability=True, random_state=41, shrinking=True,
            tol=0.001, verbose=False)
```

Figure 112: Fitting SVM classifier with best parameters from Grid Search on Shaly-limestone data

Here the "*probability*" is set to "*True*" which would make the SVM classifier give its confidence level (predict_proba) that will be useful later to plot the ROC curves.

Comparing with previous version before Grid Search:

```
In [196]: rbf_kernel_svm_clf.fit(X_train, y_train_sl)

Out[196]: SVC(C=1, cache_size=200, class_weight=None, coef0=0.0,
            decision_function_shape='ovr', degree=3, gamma='auto', kernel='rbf',
            max_iter=-1, probability=True, random_state=None, shrinking=True,
            tol=0.001, verbose=False)

In [197]: y_test_svm_noGS = rbf_kernel_svm_clf.predict(X_test_prepared)

In [198]: f1_score(y_test_sl, y_test_svm_noGS , average="weighted")

Out[198]: 0.88354820395494993

In [199]: roc_auc_score(y_test_sl, y_test_svm_noGS, average = 'weighted')

Out[199]: 0.8271113323378132
```

Figure 113: SVM scores without Grid Search on Shaly-Limestone data

With Grid Search:

```
In [192]: y_pred_svm_sl  = cross_val_predict(svm_clf_sl_GS, X_train, y_train_sl, cv=5, n_jobs=-1)
          f1_score(y_train_sl, y_pred_svm_sl , average="weighted")
Out[192]: 0.92902906979407762

In [193]: y_test_svm_pred = svm_clf_sl_GS.predict(X_test_prepared)

In [194]: f1_score(y_test_sl, y_test_svm_pred , average="weighted")
Out[194]: 0.93652014192326571

In [195]: roc_auc_score(y_test_sl, y_test_svm_pred, average = 'weighted')
Out[195]: 0.92164426746308281
```

Figure 114: Scores with Grid Search on Shaly Limestone data

With Grid Search on SVM classifier the F1-scores and ROC-AUC scores boosted up to 11%!

Finally,

RandomForest on shaly limestone grid search

```
In [202]: numEstim_for_sl = [200, 400, 500]
          criteria_for_sl = ['gini', 'entropy']

In [203]: param_grid_for_sl  = dict(n_estimators = numEstim_for_sl, criterion = criteria_for_sl)

In [204]: grid_for_sl = GridSearchCV(forest_clf_n,param_grid=param_grid_for_sl,cv=5, n_jobs =-1)

In [205]: grid_for_sl.fit(X_train, y_train_sl)
Out[205]: GridSearchCV(cv=5, error_score='raise',
                estimator=RandomForestClassifier(bootstrap=True, class_weight=None, criterion='gini',
                     max_depth=None, max_features='auto', max_leaf_nodes=None,
                     min_impurity_decrease=0.0, min_impurity_split=None,
                     min_samples_leaf=1, min_samples_split=2,
                     min_weight_fraction_leaf=0.0, n_estimators=10, n_jobs=1,
                     oob_score=False, random_state=42, verbose=0, warm_start=False),
                fit_params=None, iid=True, n_jobs=-1,
                param_grid={'n_estimators': [200, 400, 500], 'criterion': ['gini', 'entropy']},
                pre_dispatch='2*n_jobs', refit=True, return_train_score=True,
                scoring=None, verbose=0)
```

Figure 115: Random Forest Classifier Grid Search on Shaly Limestone class

```
In [206]: grid_for_sl.best_params_

Out[206]: {'criterion': 'gini', 'n_estimators': 500}

In [207]: grid_for_sl.best_estimator_

Out[207]: RandomForestClassifier(bootstrap=True, class_weight=None, criterion='gini',
                       max_depth=None, max_features='auto', max_leaf_nodes=None,
                       min_impurity_decrease=0.0, min_impurity_split=None,
                       min_samples_leaf=1, min_samples_split=2,
                       min_weight_fraction_leaf=0.0, n_estimators=500, n_jobs=1,
                       oob_score=False, random_state=42, verbose=0, warm_start=False)

In [208]: for_clf_sl_GS = RandomForestClassifier(n_estimators = 500, random_state =42)

In [209]: for_clf_sl_GS.fit(X_train, y_train_sl)

Out[209]: RandomForestClassifier(bootstrap=True, class_weight=None, criterion='gini',
                       max_depth=None, max_features='auto', max_leaf_nodes=None,
                       min_impurity_decrease=0.0, min_impurity_split=None,
                       min_samples_leaf=1, min_samples_split=2,
                       min_weight_fraction_leaf=0.0, n_estimators=500, n_jobs=1,
                       oob_score=False, random_state=42, verbose=0, warm_start=False)
```

Figure 116: Finding best parameters of Random Forest Classifier on Shaly-limestone and fitting with the data

Comparing without GridSearch:

```
In [214]: y_test_for_noGS= forest_clf_n.predict(X_test_prepared)

In [215]: f1_score(y_test_sl, y_test_for_noGS, average="weighted")

Out[215]: 0.95636735796639238

In [216]: roc_auc_score(y_test_sl, y_test_for_noGS, average = 'weighted')

Out[216]: 0.92172722747635638
```

Figure 117: Scores without Grid Search of Random Forest on Shaly-Limestone

With Grid Search:

```
In [211]: y_test_for_pred = for_clf_sl_GS.predict(X_test_prepared)

In [212]: f1_score(y_test_sl, y_test_for_pred , average="weighted")
Out[212]: 0.95460353713498058

In [213]: roc_auc_score(y_test_sl, y_test_for_pred, average = 'weighted')
Out[213]: 0.92324124771859972
```

Figure 118: Random Forest Classifier with Grid Search parameters on Shaly-Limestone class scores

The performance on the F1-score apparently decreased very slightly with Grid Search but the ROC-AUC score increased which means the classifier does better than it was without Grid Search.

3.1.2.6.5.3 : Voting on shaly-limestone:

Once we have our best form of our classifiers with Grid Search, we can now do Voting as we have learned about already earlier. The Voting classifier would boost the performance of the classifier on the training and then on test dataset. Here's the implementation:

```
In [219]: from sklearn.ensemble import VotingClassifier

          voting_clf_sl = VotingClassifier(estimators = [('svmGS', svm_clf_sl_GS),
                                                         ('knnGSsl', knn_clf_sl_GS),
                                                         ('rnfGS', for_clf_sl_GS)], voting = 'soft')

In [220]: voting_clf_sl.fit(X_train, y_train_sl)
Out[220]: VotingClassifier(estimators=[('svmGS', SVC(C=100, cache_size=200, class_weight=None, coef0=0.0,
            decision_function_shape='ovo', degree=3, gamma='auto', kernel='rbf',
            max_iter=-1, probability=True, random_state=41, shrinking=True,
            tol=0.001, verbose=False)), ('knnGSsl', KNeighborsClassifier(algorithm='auto', lea...timators=500, n_jobs=1,
                       oob_score=False, random_state=42, verbose=0, warm_start=False))],
                     flatten_transform=None, n_jobs=1, voting='soft', weights=None)

In [221]: y_pred_voting_sl  = cross_val_predict(voting_clf_sl, X_train, y_train_sl, cv=5, n_jobs=-1)
          f1_score(y_train_sl, y_pred_voting_sl , average="weighted")
Out[221]: 0.93531625139763241

In [222]: y_test_voting_pred = voting_clf_sl.predict(X_test_prepared)

In [223]: f1_score(y_test_sl, y_test_voting_pred , average="weighted")
Out[223]: 0.95138816147323446

In [224]: roc_auc_score(y_test_sl, y_test_voting_pred)
Out[224]: 0.93184834909573577
```

Figure 119: Fitting all classifiers in Voting Classifier and their scores

Again, the ROC-AUC scores boosted up further with voting b/w the three classifiers about 1%. We can now see the ROC-AUC curves on the test set of each of these individual classifiers and then voting between these classes:

```
In [227]: plt.figure(figsize=(8, 6))
          plot_roc_curve(fpr_voting_sl, tpr_voting_sl, "Voting Classifier")
          plot_roc_curve(fpr_forest_sl, tpr_forest_sl, "Random Forest")
          plot_roc_curve(fpr_svm_sl, tpr_svm_sl, "Gaussian RBF SVM Classifier")
          plot_roc_curve(fpr_knn_sl, tpr_knn_sl, "K-Nearest Neighbor")
          plt.legend(loc="lower right", fontsize=16)
          plt.title("ROC Curves on Test set of Shaly Limestone")
          plt.show()
```

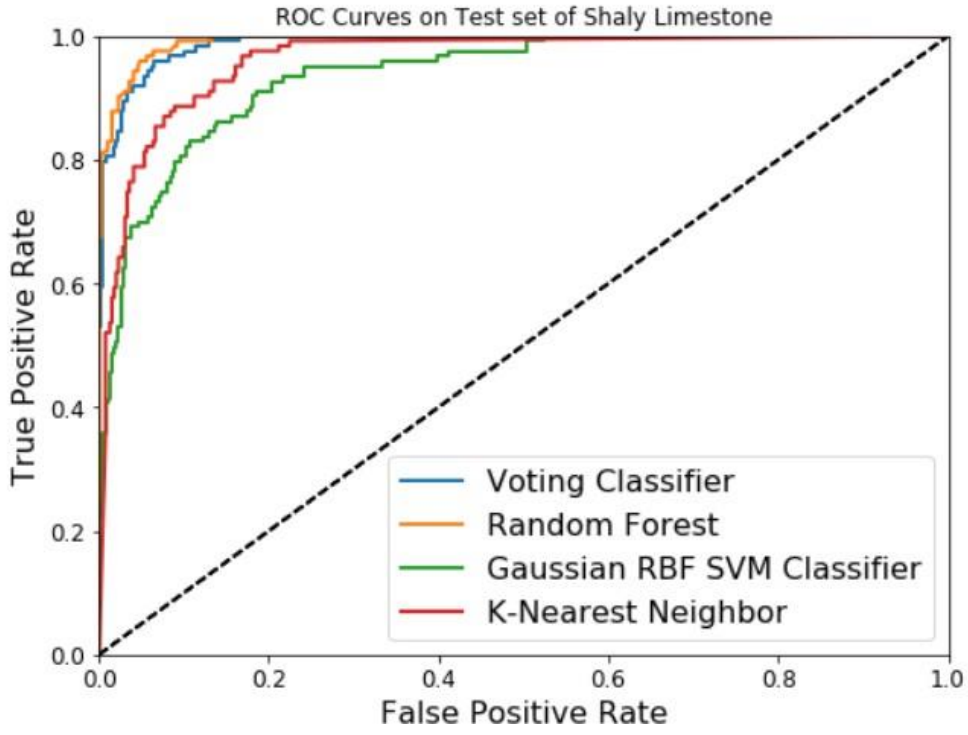

Figure 120: ROC curve of all classifiers with Grid Search on Shaly-Limestone class test set

Initially, as seen, the Random Forest does even better than the Voting classifier but eventually it would outrun the Random Forest classifier on the overall test set which can be verified by the ROC-AUC scores we saw just above.

3.1.2.6.5.3 : Generalization of all classifiers on all the classes:

With this, we can make a general code for all the classes in the training and test sets by automating it with a for loop. Below is the code demonstration for this:

```
In [277]: y_trains_classes= (y_train_sl, y_train_lim, y_train_shale, y_train_sandlim,
                             y_train_ss, y_train_dol, y_train_sand)
          y_classes_names = ("shaly limestone", "limestone", "shale", "sandy lime",
                             "shaly sandstone", "dolomite", "sandstone")
          y_test_classes = (y_test_sl, y_test_lim, y_test_shale, y_test_sandlim, y_test_ss, y_test_dol, y_test_sand)
```

Figure 121: Combining all classes to be used in for loop for generalizing

In the above code, we have put all the classes in training and test sets together and their strings that were used in printing the result for each class separately and make the algorithm ready for automation by a "*For loop*".

```
In [278]: from sklearn.svm import SVC

          svm_clf_GS = SVC(C= 100, decision_function_shape = 'ovo', random_state =41, probability = True)

          for y_train_all, y_test_all, y_strings_all in zip(y_trains_classes,
                                                            y_test_classes, y_classes_names):
              svm_clf_GS.fit(X_train, y_train_all)
              y_cv_svm = cross_val_score(svm_clf_GS, X_train, y_train_all, cv = 4)
              print("\n","cross-validation score on training set for", y_strings_all, "with SVMClassifier =", y_cv_svm )

              y_cvp_svm = cross_val_predict(svm_clf_GS, X_train, y_train_all, cv=4)
              f1_cvp_train_svm = f1_score(y_train_all, y_cvp_svm, average = 'weighted', labels = np.unique(y_cvp_svm))
              print("f1 scores of cross validation prediction training set for", y_strings_all, "with SVMClassifier =", f1_cvp_train_svm)
              roc_auc_score_svm_train = roc_auc_score(y_train_all, y_cvp_svm, average = 'weighted')
              print("roc auc score on cross validation prediction training set for",
                    y_strings_all,"with SVMClassifier =", roc_auc_score_svm_train)


              y_cvp_svm_test = svm_clf_GS.predict(X_test_prepared)
              f1_svm_test_all = f1_score(y_test_all, y_cvp_svm_test, average = 'weighted', labels = np.unique(y_cvp_svm_test))
              print("f1 score of actual prediction on test set of", y_strings_all,
                    "with SVMClassifier =", f1_svm_test_all)
              roc_auc_score_svm_test_all = roc_auc_score(y_test_all, y_cvp_svm_test, average = 'weighted')
              print("roc auc score of actual prediction on test set of",y_strings_all, "with SVMClassifier =",
                    roc_auc_score_svm_test_all)
```

Figure 122: General code for SVM classifier for all classes with Grid Search parameters

The above code is used to test SVM Classifier that we had from the Grid Search with all the instances on training and test sets, along with cross validation scores and cross validation predictions. Here's the output of the code:

```
 cross-validation score on training set for shaly limestone with SVMClassifier = [ 0.93696275  0.91930836  0.92507205  0.916426
51]
f1 scores of cross validation prediction training set for shaly limestone with SVMClassifier = 0.924499183061
roc auc score on cross validation prediction training set for shaly limestone with SVMClassifier = 0.896850966387
f1 score of actual prediction on test set of shaly limestone with SVMClassifier = 0.936520141923
roc auc score of actual prediction on test set of shaly limestone with SVMClassifier = 0.921644267463

 cross-validation score on training set for limestone with SVMClassifier = [ 0.91091954  0.89942529  0.93371758  0.91642651]
f1 scores of cross validation prediction training set for limestone with SVMClassifier = 0.915092443713
roc auc score on cross validation prediction training set for limestone with SVMClassifier = 0.915056644086
f1 score of actual prediction on test set of limestone with SVMClassifier = 0.939770401009
roc auc score of actual prediction on test set of limestone with SVMClassifier = 0.939854613459

 cross-validation score on training set for shale with SVMClassifier = [ 0.93965517  0.90804598  0.92507205  0.93948127]
f1 scores of cross validation prediction training set for shale with SVMClassifier = 0.926516278593
roc auc score on cross validation prediction training set for shale with SVMClassifier = 0.826030197897
f1 score of actual prediction on test set of shale with SVMClassifier = 0.952510764485
roc auc score of actual prediction on test set of shale with SVMClassifier = 0.882744783307

 cross-validation score on training set for sandy lime with SVMClassifier = [ 1.          0.99712644  1.          1.        ]
f1 scores of cross validation prediction training set for sandy lime with SVMClassifier = 0.999274167322
roc auc score on cross validation prediction training set for sandy lime with SVMClassifier = 0.982142857143
f1 score of actual prediction on test set of sandy lime with SVMClassifier = 0.996116504854
roc auc score of actual prediction on test set of sandy lime with SVMClassifier = 0.94900990099


 cross-validation score on training set for shaly sandstone with SVMClassifier = [ 0.99712644  1.          1.          1.
  ]
f1 scores of cross validation prediction training set for shaly sandstone with SVMClassifier = 0.999229317798
roc auc score on cross validation prediction training set for shaly sandstone with SVMClassifier = 0.875
f1 score of actual prediction on test set of shaly sandstone with SVMClassifier = 1.0
roc auc score of actual prediction on test set of shaly sandstone with SVMClassifier = 1.0

 cross-validation score on training set for dolomite with SVMClassifier = [ 1.          1.          1.          0.99711816]
f1 scores of cross validation prediction training set for dolomite with SVMClassifier = 0.999278831168
roc auc score on cross validation prediction training set for dolomite with SVMClassifier = 0.994791666667
f1 score of actual prediction on test set of dolomite with SVMClassifier = 1.0
roc auc score of actual prediction on test set of dolomite with SVMClassifier = 1.0

 cross-validation score on training set for sandstone with SVMClassifier = [ 1.  1.  1.  1.]
f1 scores of cross validation prediction training set for sandstone with SVMClassifier = 1.0
roc auc score on cross validation prediction training set for sandstone with SVMClassifier = 1.0
f1 score of actual prediction on test set of sandstone with SVMClassifier = 0.996226826208
roc auc score of actual prediction on test set of sandstone with SVMClassifier = 0.997995991984
```

Figure 123: Scores for SVM classifier for all classes with Grid Search parameters

Similarly, we can see the performance of Random Forest, K-Nearest Neighbor and Voting classifier on the rest of the classes too.

In [279]:
```
from sklearn.neighbors import KNeighborsClassifier

knn_clf_GS = KNeighborsClassifier(n_neighbors = 5, weights = 'distance')

for y_train_all, y_test_all, y_strings_all in zip(y_trains_classes,
                                                  y_test_classes, y_classes_names):
    knn_clf_GS.fit(X_train, y_train_all)
    y_cv_knn = cross_val_score(knn_clf_GS, X_train, y_train_all, cv = 4)
    print("\n","cross-validation score on training set for", y_strings_all, "with KNeighbor Classifier =", y_cv_knn)

    y_cvp_knn = cross_val_predict(knn_clf_GS, X_train, y_train_all, cv=4)
    f1_cvp_train_knn = f1_score(y_train_all, y_cvp_knn, average = 'weighted', labels = np.unique(y_cvp_knn))
    print("f1 scores of cross validation prediction training set for", y_strings_all,
          "with KNeighbor Classifier =", f1_cvp_train_knn)
    roc_auc_score_knn_train = roc_auc_score(y_train_all, y_cvp_knn, average = 'weighted')
    print("roc auc score on cross validation prediction training set for",
          y_strings_all,"with KNeighbors Classifier =", roc_auc_score_knn_train)


    y_cvp_knn_test = knn_clf_GS.predict(X_test_prepared)
    f1_knn_test_all = f1_score(y_test_all, y_cvp_knn_test, average = 'weighted', labels = np.unique(y_cvp_knn_test))
    print("f1 score of actual prediction on test set of", y_strings_all,
          "with KNeighbors Classifier =", f1_knn_test_all)
    roc_auc_score_knn_test_all = roc_auc_score(y_test_all, y_cvp_knn_test, average = 'weighted')
    print("roc auc score of actual prediction on test set of",y_strings_all, "with KNeighbors Classifier =",
          roc_auc_score_knn_test_all)
```

Figure 124: General code for K-Nearest Neighbor classifier for all classes with Grid Search parameters

```
 cross-validation score on training set for shaly limestone with KNeighbor Classifier = [ 0.91404011  0.88760807  0.91066282
0.88184438]
f1 scores of cross validation prediction training set for shaly limestone with KNeighbor Classifier = 0.897175808047
roc auc score on cross validation prediction training set for shaly limestone with KNeighbors Classifier = 0.847939235356
f1 score of actual prediction on test set of shaly limestone with KNeighbors Classifier = 0.910925554061
roc auc score of actual prediction on test set of shaly limestone with KNeighbors Classifier = 0.879956860793

 cross-validation score on training set for limestone with KNeighbor Classifier = [ 0.90517241  0.88505747  0.91066282  0.88472
622]
f1 scores of cross validation prediction training set for limestone with KNeighbor Classifier = 0.896374986099
roc auc score on cross validation prediction training set for limestone with KNeighbors Classifier = 0.896338837637
f1 score of actual prediction on test set of limestone with KNeighbors Classifier = 0.908737864078
roc auc score of actual prediction on test set of limestone with KNeighbors Classifier = 0.908741290381

 cross-validation score on training set for shale with KNeighbor Classifier = [ 0.92816092  0.90229885  0.90778098  0.90201729]
f1 scores of cross validation prediction training set for shale with KNeighbor Classifier = 0.903373527514
roc auc score on cross validation prediction training set for shale with KNeighbors Classifier = 0.746079229653
f1 score of actual prediction on test set of shale with KNeighbors Classifier = 0.913566048202
roc auc score of actual prediction on test set of shale with KNeighbors Classifier = 0.761316211878

 cross-validation score on training set for sandy lime with KNeighbor Classifier = [ 1.          0.99712644  1.          1.
    ]
f1 scores of cross validation prediction training set for sandy lime with KNeighbor Classifier = 0.999274167322
roc auc score on cross validation prediction training set for sandy lime with KNeighbors Classifier = 0.982142857143
f1 score of actual prediction on test set of sandy lime with KNeighbors Classifier = 0.998008114117
roc auc score of actual prediction on test set of sandy lime with KNeighbors Classifier = 0.95


 cross-validation score on training set for shaly sandstone with KNeighbor Classifier = [ 0.99712644  1.          1.
1.        ]
f1 scores of cross validation prediction training set for shaly sandstone with KNeighbor Classifier = 0.999229317798
roc auc score on cross validation prediction training set for shaly sandstone with KNeighbors Classifier = 0.875
f1 score of actual prediction on test set of shaly sandstone with KNeighbors Classifier = 0.997735573181
roc auc score of actual prediction on test set of shaly sandstone with KNeighbors Classifier = 0.75

 cross-validation score on training set for dolomite with KNeighbor Classifier = [ 1.          1.          1.          0.997118
16]
f1 scores of cross validation prediction training set for dolomite with KNeighbor Classifier = 0.999282300287
roc auc score on cross validation prediction training set for dolomite with KNeighbors Classifier = 0.999613601236
f1 score of actual prediction on test set of dolomite with KNeighbors Classifier = 1.0
roc auc score of actual prediction on test set of dolomite with KNeighbors Classifier = 1.0

 cross-validation score on training set for sandstone with KNeighbor Classifier = [ 0.99712644  0.99712644  1.          1.
   ]
f1 scores of cross validation prediction training set for sandstone with KNeighbor Classifier = 0.998576967144
roc auc score on cross validation prediction training set for sandstone with KNeighbors Classifier = 0.999257609503
f1 score of actual prediction on test set of sandstone with KNeighbors Classifier = 0.99808669905
roc auc score of actual prediction on test set of sandstone with KNeighbors Classifier = 0.998997995992
```

Figure 125: Scores for K-Nearest Neighbors classifier for all classes with Grid Search parameters

```
In [280]: from sklearn.ensemble import RandomForestClassifier

for_clf_GS = RandomForestClassifier(n_estimators = 500, random_state =42)

for y_train_all, y_test_all, y_strings_all in zip(y_trains_classes,
                                                 y_test_classes, y_classes_names):
    for_clf_GS.fit(X_train, y_train_all)
    y_cv_randfor = cross_val_score(for_clf_GS, X_train, y_train_all, cv = 4)
    print("\n","cross-validation score on training set for", y_strings_all, "with RandomForestClassifier =", y_cv_randfor )

    y_cvp_randfor = cross_val_predict(for_clf_GS, X_train, y_train_all, cv=4)
    f1_cvp_train = f1_score(y_train_all, y_cvp_randfor, average = 'weighted', labels = np.unique(y_cvp_randfor))
    print("f1 scores of cross validation prediction training set for", y_strings_all, "=", f1_cvp_train)
    roc_auc_score_randfor_train = roc_auc_score(y_train_all, y_cvp_randfor, average = 'weighted')
    print("roc auc score on cross validation prediction training set for",
          y_strings_all,"with RandomForestClassifier =", roc_auc_score_randfor_train)


    y_cvp_randfor_test = for_clf_GS.predict(X_test_prepared)
    f1_for_test_all = f1_score(y_test_all, y_cvp_randfor_test, average = 'weighted', labels = np.unique(y_cvp_randfor_test))
    print("f1 score of actual prediction on test set of", y_strings_all,
          "with RandomForestClassifier =", f1_for_test_all)
    roc_auc_score_randfor_test_all = roc_auc_score(y_test_all, y_cvp_randfor_test, average = 'weighted')
    print("roc auc score of actual prediction on test set of",y_strings_all, "with RandomForestClassifier =",
          roc_auc_score_randfor_test_all)
```

Figure 126: General code for Random Forest classifier for all classes with Grid Search parameters

```
 cross-validation score on training set for shaly limestone with RandomForestClassifier = [ 0.95988539  0.93659942  0.94524496
0.93948127]
f1 scores of cross validation prediction training set for shaly limestone = 0.944084567529
roc auc score on cross validation prediction training set for shaly limestone with RandomForestClassifier = 0.904398248769
f1 score of actual prediction on test set of shaly limestone with RandomForestClassifier = 0.954603537135
roc auc score of actual prediction on test set of shaly limestone with RandomForestClassifier = 0.923241247719

 cross-validation score on training set for limestone with RandomForestClassifier = [ 0.93390805  0.91666667  0.94524496  0.916
42651]
f1 scores of cross validation prediction training set for limestone = 0.928053383173
roc auc score on cross validation prediction training set for limestone with RandomForestClassifier = 0.928031369561
f1 score of actual prediction on test set of limestone with RandomForestClassifier = 0.961165048544
roc auc score of actual prediction on test set of limestone with RandomForestClassifier = 0.96116490212

 cross-validation score on training set for shale with RandomForestClassifier = [ 0.93965517  0.90517241  0.92507205  0.9077809
8]
f1 scores of cross validation prediction training set for shale = 0.910755728115
roc auc score on cross validation prediction training set for shale with RandomForestClassifier = 0.744756080292
f1 score of actual prediction on test set of shale with RandomForestClassifier = 0.931006964187
roc auc score of actual prediction on test set of shale with RandomForestClassifier = 0.800401284109

 cross-validation score on training set for sandy lime with RandomForestClassifier = [ 0.99137931  0.99712644  1.          0.99
135447]
f1 scores of cross validation prediction training set for sandy lime = 0.994610770556
roc auc score on cross validation prediction training set for sandy lime with RandomForestClassifier = 0.875
f1 score of actual prediction on test set of sandy lime with RandomForestClassifier = 0.998008114117
roc auc score of actual prediction on test set of sandy lime with RandomForestClassifier = 0.95


 cross-validation score on training set for shaly sandstone with RandomForestClassifier = [ 0.99712644  0.99712644  0.99711816
0.99711816]
f1 scores of cross validation prediction training set for shaly sandstone = 0.99855907781
roc auc score on cross validation prediction training set for shaly sandstone with RandomForestClassifier = 0.5
f1 score of actual prediction on test set of shaly sandstone with RandomForestClassifier = 0.998054474708
roc auc score of actual prediction on test set of shaly sandstone with RandomForestClassifier = 0.5

 cross-validation score on training set for dolomite with RandomForestClassifier = [ 1.  1.  1.  1.]
f1 scores of cross validation prediction training set for dolomite = 1.0
roc auc score on cross validation prediction training set for dolomite with RandomForestClassifier = 1.0
f1 score of actual prediction on test set of dolomite with RandomForestClassifier = 0.998070537571
roc auc score of actual prediction on test set of dolomite with RandomForestClassifier = 0.998956158664

 cross-validation score on training set for sandstone with RandomForestClassifier = [ 0.99712644  1.          1.          0.997
10983]
f1 scores of cross validation prediction training set for sandstone = 0.998544555623
roc auc score on cross validation prediction training set for sandstone with RandomForestClassifier = 0.976744186047
f1 score of actual prediction on test set of sandstone with RandomForestClassifier = 0.99808669905
roc auc score of actual prediction on test set of sandstone with RandomForestClassifier = 0.998997995992
```

Figure 127: Scores for Random Forest classifier for all classes with Grid Search parameters

In [281]:
```
from sklearn.ensemble import VotingClassifier

voting_clf_all = VotingClassifier(estimators = [('knnGS', knn_clf_GS),
                                                ('svmGS', svm_clf_GS),
                                               ('rnfGS', for_clf_GS)], voting = 'soft')

for y_train_all, y_test_all, y_strings_all in zip(y_trains_classes,
                                                  y_test_classes, y_classes_names):

    voting_clf_all.fit(X_train, y_train_all)
    y_cv_voting = cross_val_score(voting_clf_all, X_train, y_train_all, cv = 4)
    print("\n","cross-validation score on training set for", y_strings_all,
          "with Voting b/w three classifier =", y_cv_voting )

    y_pred_voting  = cross_val_predict(voting_clf_all, X_train, y_train_all, cv=4, n_jobs=-1)
    f1_cvp_train_voting = f1_score(y_train_all, y_pred_voting , average="weighted", labels = np.unique(y_pred_voting))
    print("f1 scores of cross validation prediction training set for", y_strings_all,
          "with Voting b/w three classifier =", f1_cvp_train_voting)
    roc_auc_score_voting_train = roc_auc_score(y_train_all, y_pred_voting, average = 'weighted')
    print("roc auc score on cross validation prediction training set for"
          , y_strings_all,"with Voting b/w three classifier =", roc_auc_score_voting_train)

    y_test_voting_pred_all = voting_clf_all.predict(X_test_prepared)
    f1_test_voting_all = f1_score(y_test_all, y_test_voting_pred_all,
                                  average= 'weighted', labels = np.unique(y_test_voting_pred_all))
    print("f1 score of actual prediction on test set of", y_strings_all,
          "with Voting b/w three classifier =", f1_test_voting_all)
    roc_auc_score_voting_test_all = roc_auc_score(y_test_all, y_test_voting_pred_all, average= 'weighted')
    print("roc auc score of actual prediction on test set of",y_strings_all,
          "with Voting b/w three classifier =", roc_auc_score_voting_test_all)
```

Figure 128: General code for Voting classifier with combined classifiers on all classes of the data set

```
 cross-validation score on training set for shaly limestone with Voting b/w three classifier = [ 0.94269341  0.93659942  0.9365
9942  0.93371758]
f1 scores of cross validation prediction training set for shaly limestone with Voting b/w three classifier = 0.936334567828
roc auc score on cross validation prediction training set for shaly limestone with Voting b/w three classifier = 0.898166378299
f1 score of actual prediction on test set of shaly limestone with Voting b/w three classifier = 0.951388161473
roc auc score of actual prediction on test set of shaly limestone with Voting b/w three classifier = 0.931848349096

 cross-validation score on training set for limestone with Voting b/w three classifier = [ 0.93103448  0.90517241  0.94524496
0.93083573]
f1 scores of cross validation prediction training set for limestone with Voting b/w three classifier = 0.928053383173
roc auc score on cross validation prediction training set for limestone with Voting b/w three classifier = 0.928031369561
f1 score of actual prediction on test set of limestone with Voting b/w three classifier = 0.953395949694
roc auc score of actual prediction on test set of limestone with Voting b/w three classifier = 0.953412964136

 cross-validation score on training set for shale with Voting b/w three classifier = [ 0.93390805  0.90804598  0.92795389  0.91
642651]
f1 scores of cross validation prediction training set for shale with Voting b/w three classifier = 0.912999364118
roc auc score on cross validation prediction training set for shale with Voting b/w three classifier = 0.748247601515
f1 score of actual prediction on test set of shale with Voting b/w three classifier = 0.941697784416
roc auc score of actual prediction on test set of shale with Voting b/w three classifier = 0.824077046549

 cross-validation score on training set for sandy lime with Voting b/w three classifier = [ 1.          0.99712644  1.
1.        ]
f1 scores of cross validation prediction training set for sandy lime with Voting b/w three classifier = 0.999274167322
roc auc score on cross validation prediction training set for sandy lime with Voting b/w three classifier = 0.982142857143
f1 score of actual prediction on test set of sandy lime with Voting b/w three classifier = 0.998008114117
roc auc score of actual prediction on test set of sandy lime with Voting b/w three classifier = 0.95


 cross-validation score on training set for shaly sandstone with Voting b/w three classifier = [ 0.99712644  0.99712644  1.
     0.99711816]
f1 scores of cross validation prediction training set for shaly sandstone with Voting b/w three classifier = 0.997195411239
roc auc score on cross validation prediction training set for shaly sandstone with Voting b/w three classifier = 0.625
f1 score of actual prediction on test set of shaly sandstone with Voting b/w three classifier = 0.998054474708
roc auc score of actual prediction on test set of shaly sandstone with Voting b/w three classifier = 0.5

 cross-validation score on training set for dolomite with Voting b/w three classifier = [ 1.  1.  1.  1.]
f1 scores of cross validation prediction training set for dolomite with Voting b/w three classifier = 1.0
roc auc score on cross validation prediction training set for dolomite with Voting b/w three classifier = 1.0
f1 score of actual prediction on test set of dolomite with Voting b/w three classifier = 1.0
roc auc score of actual prediction on test set of dolomite with Voting b/w three classifier = 1.0

 cross-validation score on training set for sandstone with Voting b/w three classifier = [ 1.  1.  1.  1.]
f1 scores of cross validation prediction training set for sandstone with Voting b/w three classifier = 1.0
roc auc score on cross validation prediction training set for sandstone with Voting b/w three classifier = 1.0
f1 score of actual prediction on test set of sandstone with Voting b/w three classifier = 0.99808669905
roc auc score of actual prediction on test set of sandstone with Voting b/w three classifier = 0.998997995992
```

Figure 129: Scores with Voting Classifier on all classes of dataset

With that we can visualize the performance on each class by the ROC-curves first on the training set, followed by the performance on test set. Here's the ROC-curves code and plots on the training set:

```
for y_train_all, y_test_all, y_strings_all in zip(y_trains_classes,
                                                  y_test_classes, y_classes_names):
    #ROC-curves of SVM
    y_svm_proba_train = cross_val_predict(svm_clf_GS, X_train, y_train_all, cv=4,
                            method="predict_proba")
    y_svm_scores_train = y_svm_proba_train[:, 1]

    fpr_svm_train, tpr_svm_train, thresholds_svm_train = roc_curve(
                                                                    y_train_all,
                                                                    y_svm_scores_train)


    #ROC-curves for KNN
    y_knn_proba_train = cross_val_predict(knn_clf_GS, X_train, y_train_all, cv=4,
                            method="predict_proba")
    y_knn_scores_train = y_knn_proba_train[:, 1]

    fpr_knn_train, tpr_knn_train, thresholds_knn_train = roc_curve(
                                                                    y_train_all,
                                                                    y_knn_scores_train)


    #ROC-curves for RandomForestClf
    y_randfor_proba_train = cross_val_predict(for_clf_GS, X_train, y_train_all, cv=4,
                            method="predict_proba")
    y_randfor_scores_train = y_randfor_proba_train[:, 1]

    fpr_randfor_train, tpr_randfor_train, thresholds_randfor_trainr = roc_curve(
                                                                        y_train_all,
                                                                        y_randfor_scores_train)


    #ROC-curves for voting

    y_voting_proba_train_all = cross_val_predict(voting_clf_all, X_train, y_train_all, cv=4,
                            method="predict_proba")
    y_voting_scores_train_all = y_voting_proba_train_all[:, 1]
    fpr_voting_train_all, tpr_voting_train_all, thresholds_voting_train_all = roc_curve(y_train_all,
                                                                         y_voting_scores_train_all)

    #Plotting ROC-curves for each class
    plt.figure(figsize=(8, 6))
    plot_roc_curve(fpr_svm_train, tpr_svm_train, "SVM Classifier")
    plot_roc_curve(fpr_knn_train, tpr_knn_train, "K-Nearest Neighbor")
    plot_roc_curve(fpr_randfor_train, tpr_randfor_train, "Random Forest")
    plot_roc_curve(fpr_voting_train_all, tpr_voting_train_all, "Voting Classifier")
    plt.legend(loc="lower right", fontsize=16)
    plt.title('ROC curve on training set of  %s'%(y_strings_all))
    plt.axis([-0.01, 1.01, -0.01, 1.01])
    plt.show()
```

Figure 130: General code for plotting ROC curve for all classes of training data

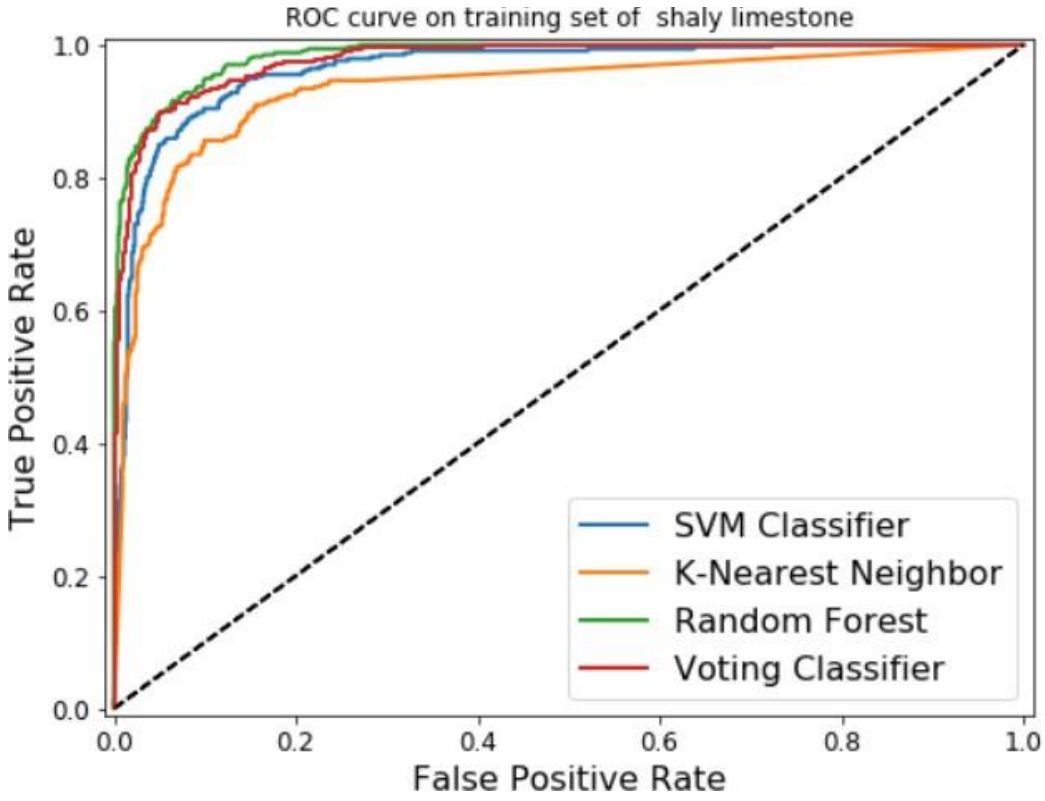

Figure 131: ROC curve of training set on Shaly Limestone class as seen before

It's clearly visible that the Random Forest Classifier outruns all other classifier in Shaly-Limestone formation. It might not make sense that Voting classifier doesn't perform better than Random Forest, but it is expected that the Voting classifier wouldn't do well in case when it has bad input classifiers. Although, K-Nearest Neighbor classifier and SVM Classifier does well on the training sets individually, but when combined with the best classifier among the three i.e., Random Forest, it might result in lowering of performance of voting classifier.

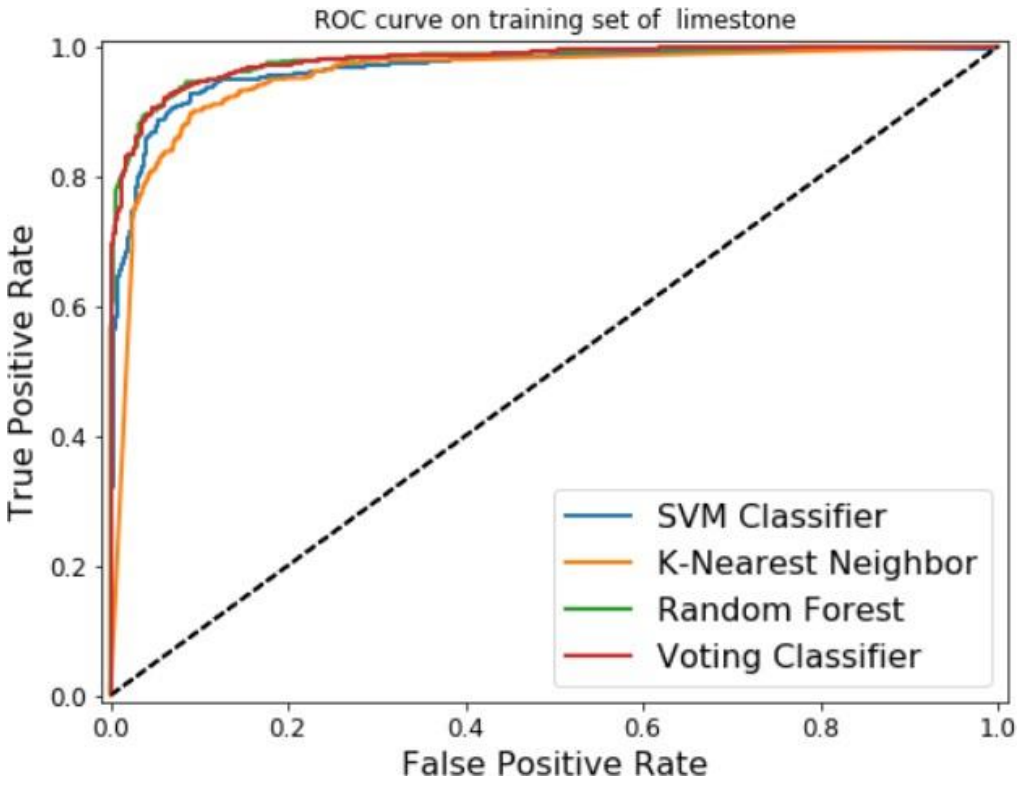

Figure 132: ROC curve of all classifiers with training on Limestone

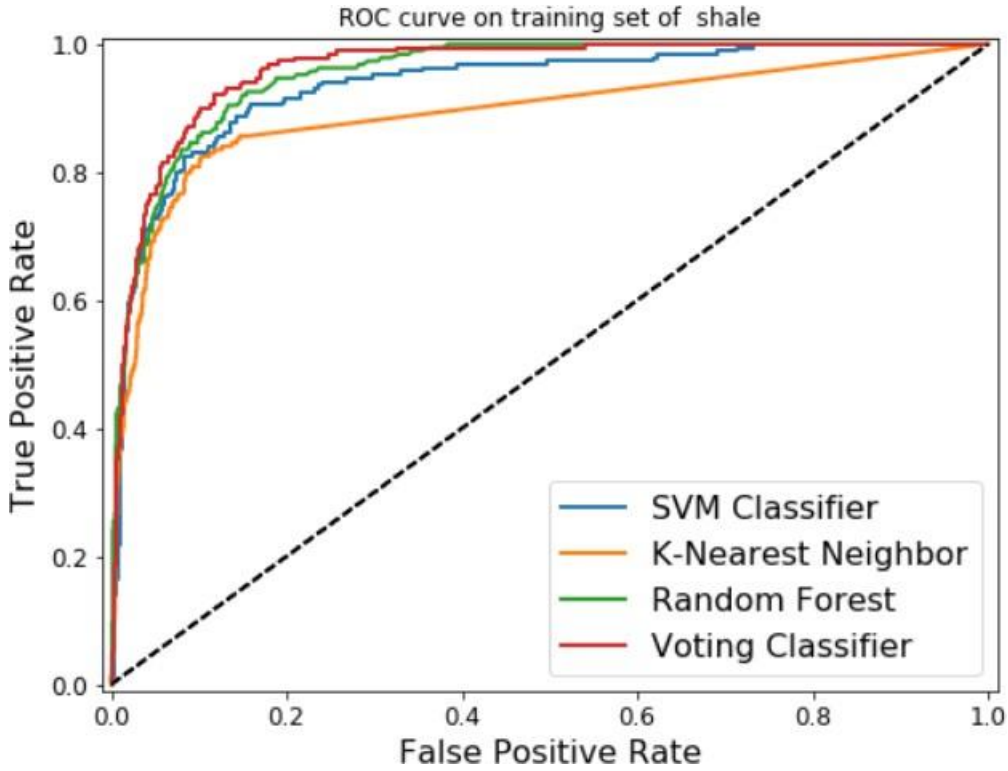

Figure 133: ROC curve of all classifiers with training on Shale

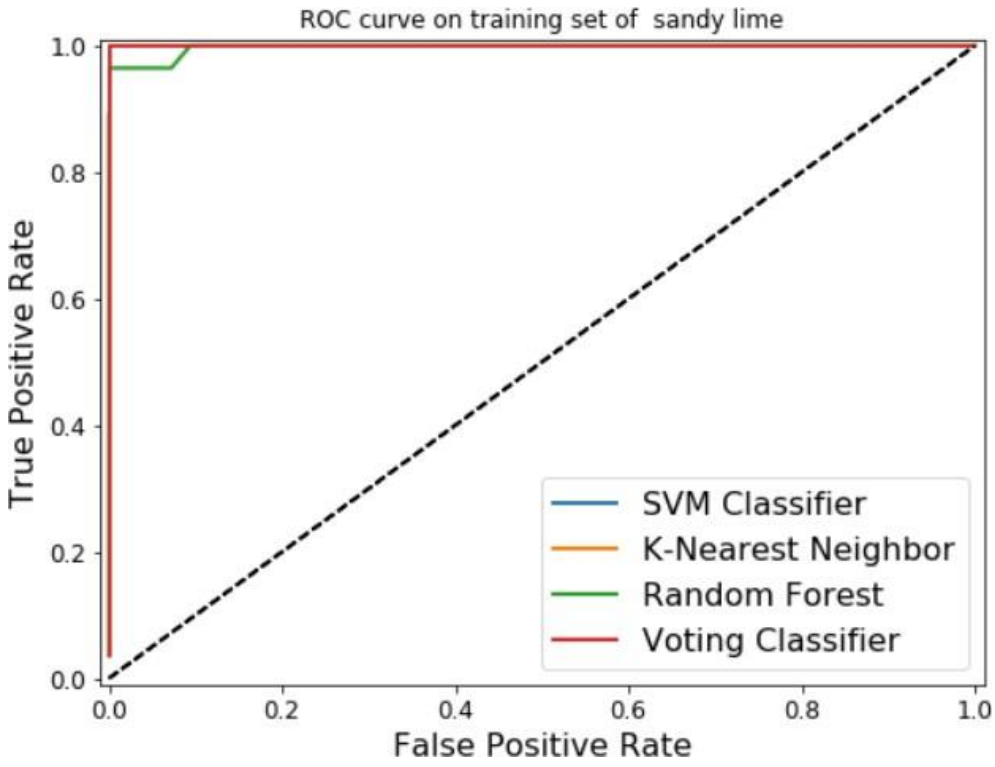

Figure 134: ROC curve of all classifiers with training on Sandy-Lime

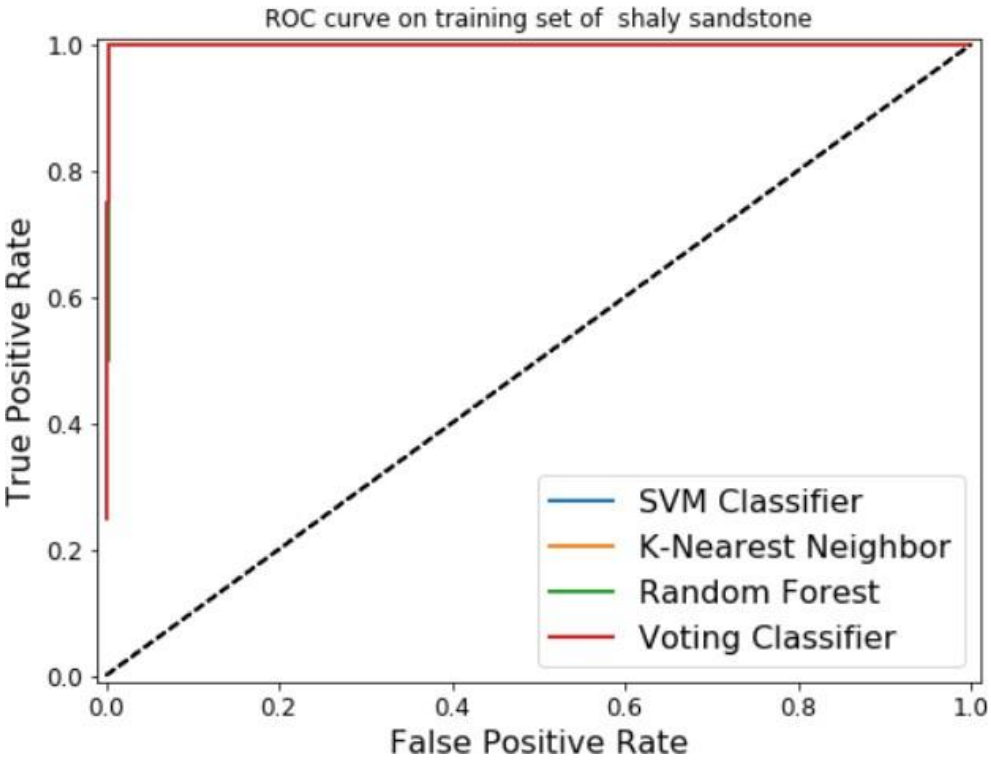

Figure 135: ROC curve of all classifiers with training on Shaly-Sandstone

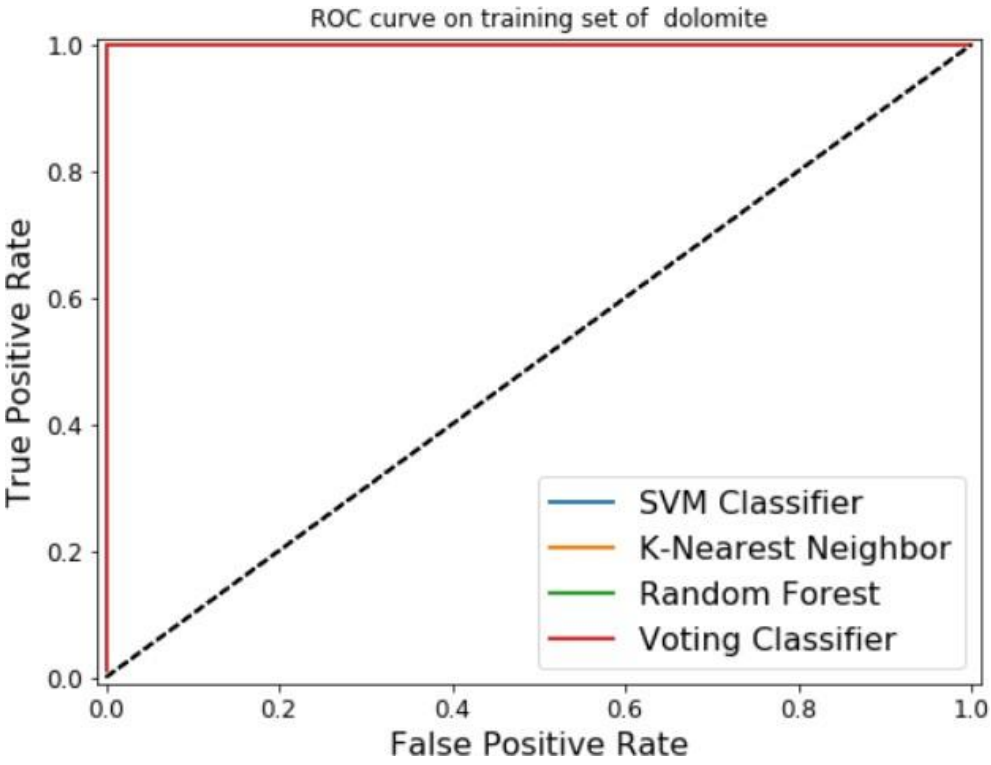

Figure 136: ROC curve of all classifiers with training on Dolomite

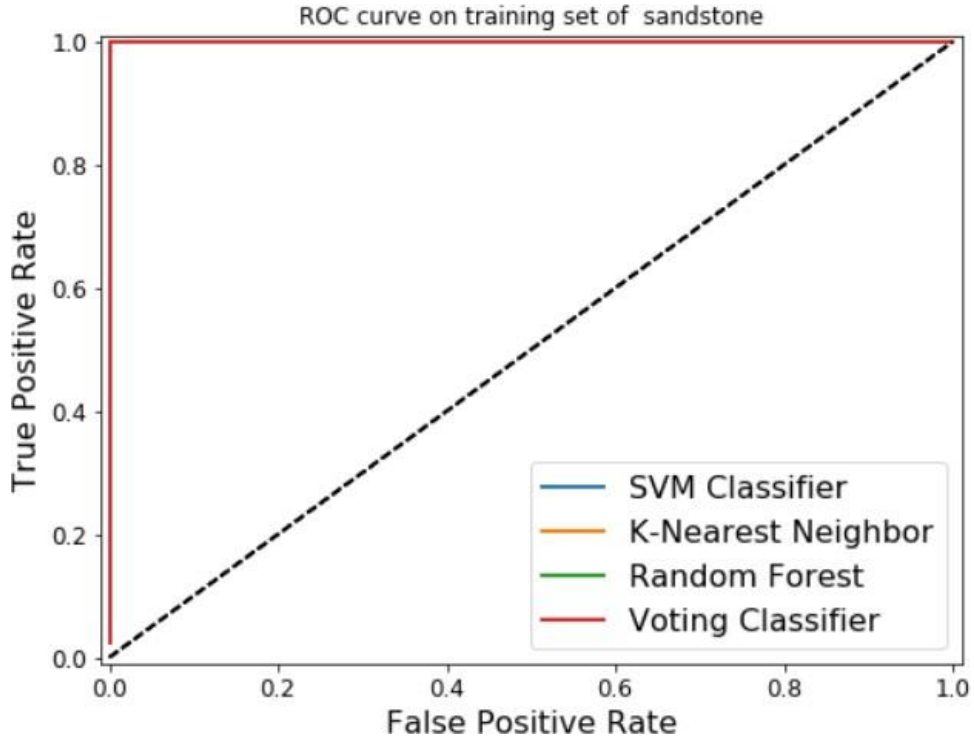

Figure 137: ROC curve of all classifiers with training on Sandstone

Note that in the last few classes, all classifier seems to have a perfect ROC-curve, with an ROC score of almost 1 for all the instances. This is because we had a few instances to fit on the training data and then even fewer on the test set, which is why the performance of the classifiers is almost perfect always. You would see the same behavior on the test set for these classes as well.

Also, the Voting Classifier except for shaly limestone class does the best on all the classes like it was expected. The explanation for its odd performance on shaly limestone class has been explained just above.

Now to see the performance on the test set:

```
for y_train_all, y_test_all, y_strings_all in zip(y_trains_classes,
                                                  y_test_classes, y_classes_names):

    svm_clf_GS.fit(X_train, y_train_all)
    y_svm_proba_test = svm_clf_GS.predict_proba(X_test_prepared)
    y_svm_scores_test = y_svm_proba_test[:, 1]

    fpr_svm_test, tpr_svm_test, thresholds_svm_test = roc_curve(y_test_all,
                                                                y_svm_scores_test)

    knn_clf_GS.fit(X_train, y_train_all)
    y_knn_proba_test = knn_clf_GS.predict_proba(X_test_prepared)
    y_knn_scores_test = y_knn_proba_test[:, 1]

    fpr_knn_test, tpr_knn_test, thresholds_knn_test = roc_curve(y_test_all,
                                                                y_knn_scores_test)

    for_clf_GS.fit(X_train, y_train_all)
    y_randfor_proba_test = for_clf_GS.predict_proba(X_test_prepared)
    y_randfor_scores_test = y_randfor_proba_test[:, 1]

    fpr_randfor_test, tpr_randfor_test, thresholds_randfor_test = roc_curve(y_test_all,
                                                                            y_randfor_scores_test)

    voting_clf_all.fit(X_train, y_train_all)
    y_probas_voting_all = voting_clf_all.predict_proba(X_test_prepared)
    y_scores_voting_all = y_probas_voting_all[:, 1] # score = proba of positive class
    fpr_voting_test_all, tpr_voting_test_all, thresholds_voting_test_all = roc_curve(y_test_all,
                                                                                     y_scores_voting_all)

    plt.figure(figsize=(8, 6))
    plot_roc_curve(fpr_svm_test, tpr_svm_test, "SVM Classifier")
    plot_roc_curve(fpr_knn_test, tpr_knn_test, "K-Nearest Neighbor")
    plot_roc_curve(fpr_randfor_test, tpr_randfor_test, "Random Forest")
    plot_roc_curve(fpr_voting_test_all, tpr_voting_test_all, "Voting Classifier")
    plt.legend(loc="lower right", fontsize=16)
    plt.title('ROC curve on test of  %s'%(y_strings_all))
    plt.axis([-0.01, 1.01, -0.01, 1.01])
    plt.show()
```

Figure 138: General code for ROC curve generation for all classes of test set

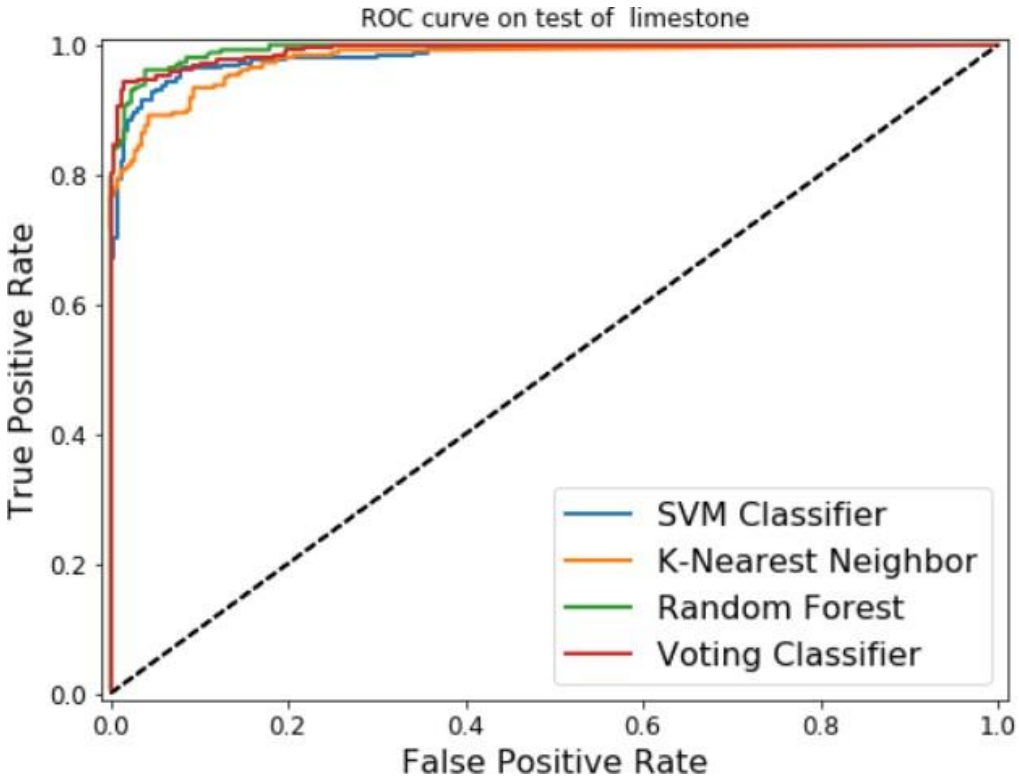

Figure 139: : ROC curve of all classifiers on test-set of Limestone

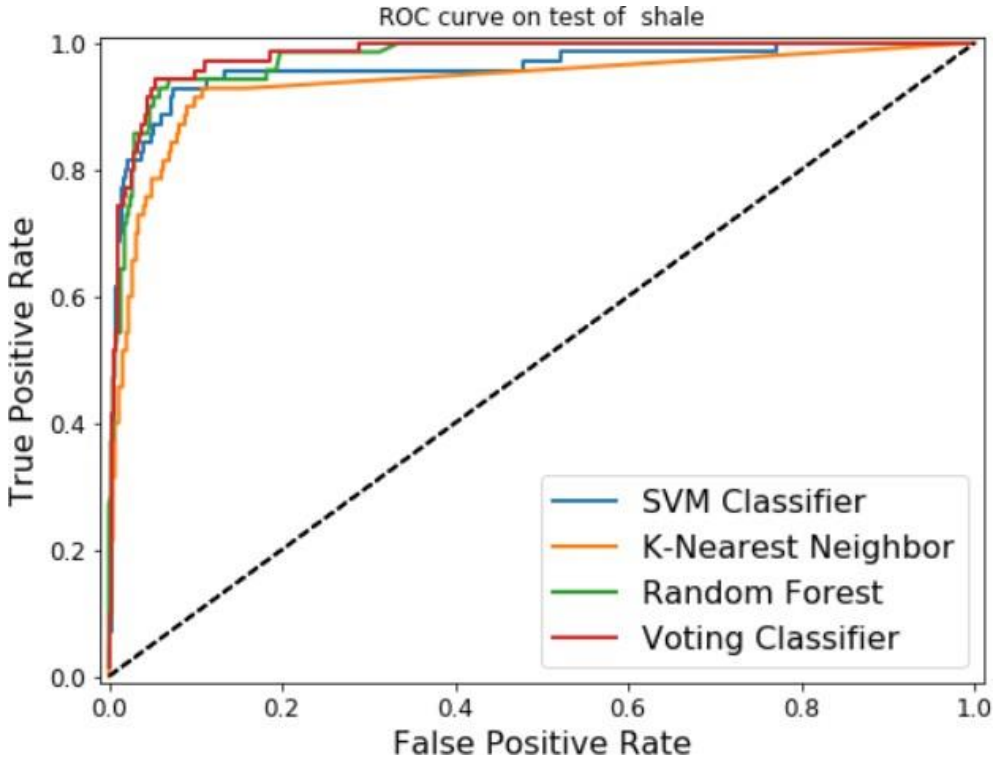

Figure 140: ROC curve of all classifiers on test-set of Shale

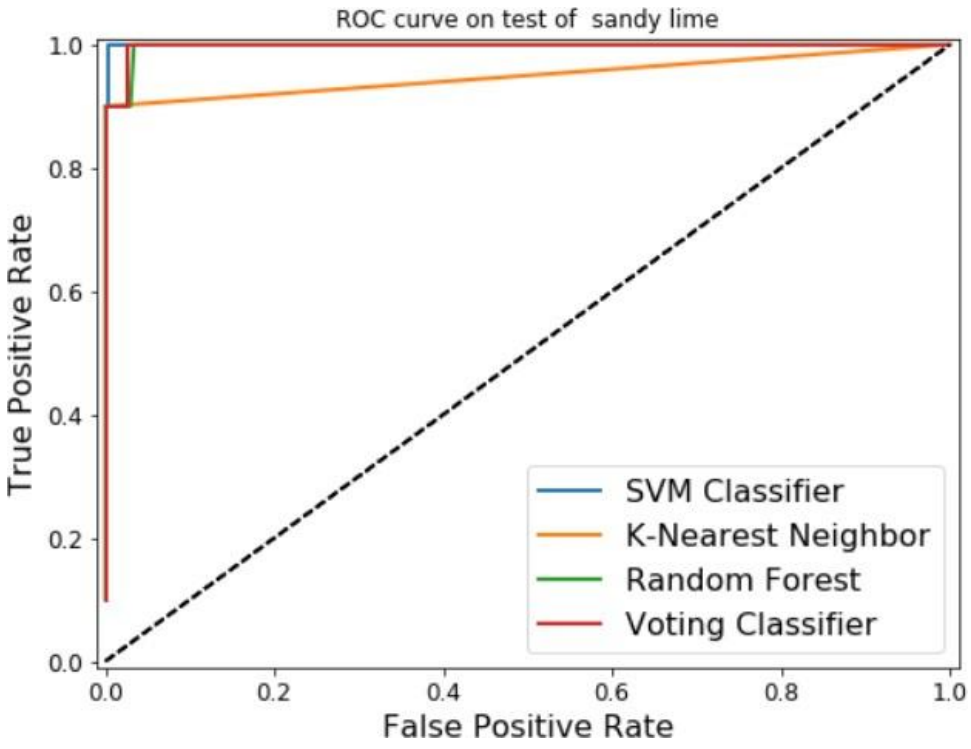

Figure 141: ROC curve of all classifiers on test-set of Sandy-Lime

Interestingly, on the test set, SVM Classifier seems to have outrun the Voting Classifier.

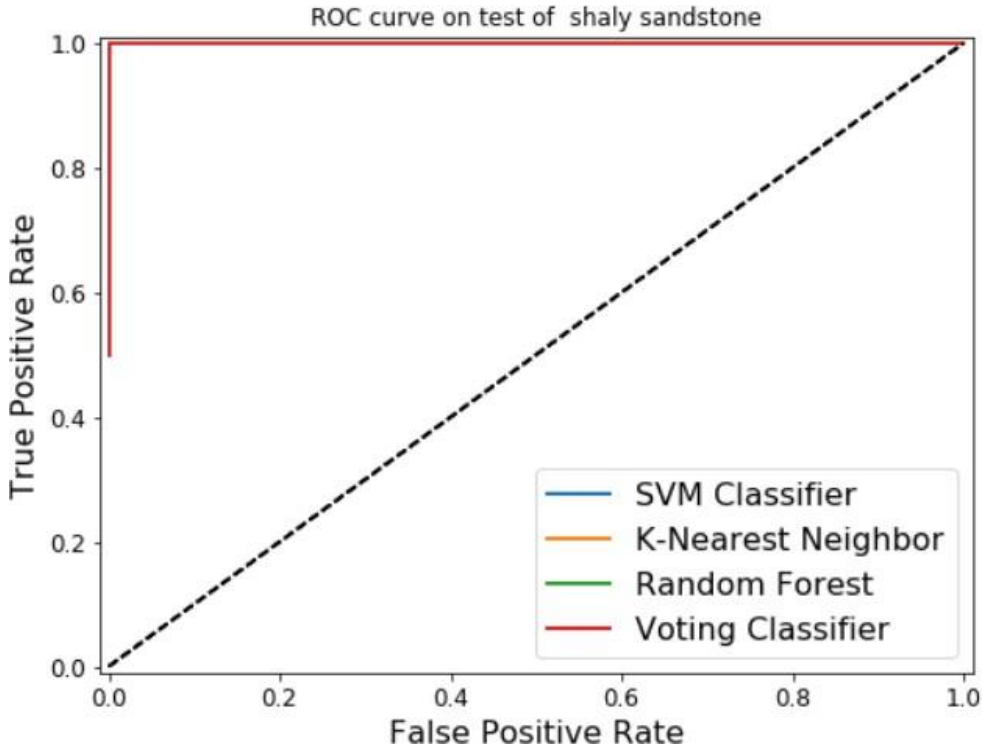

Figure 142: ROC curve of all classifiers on test-set of Shaly-Sandstone

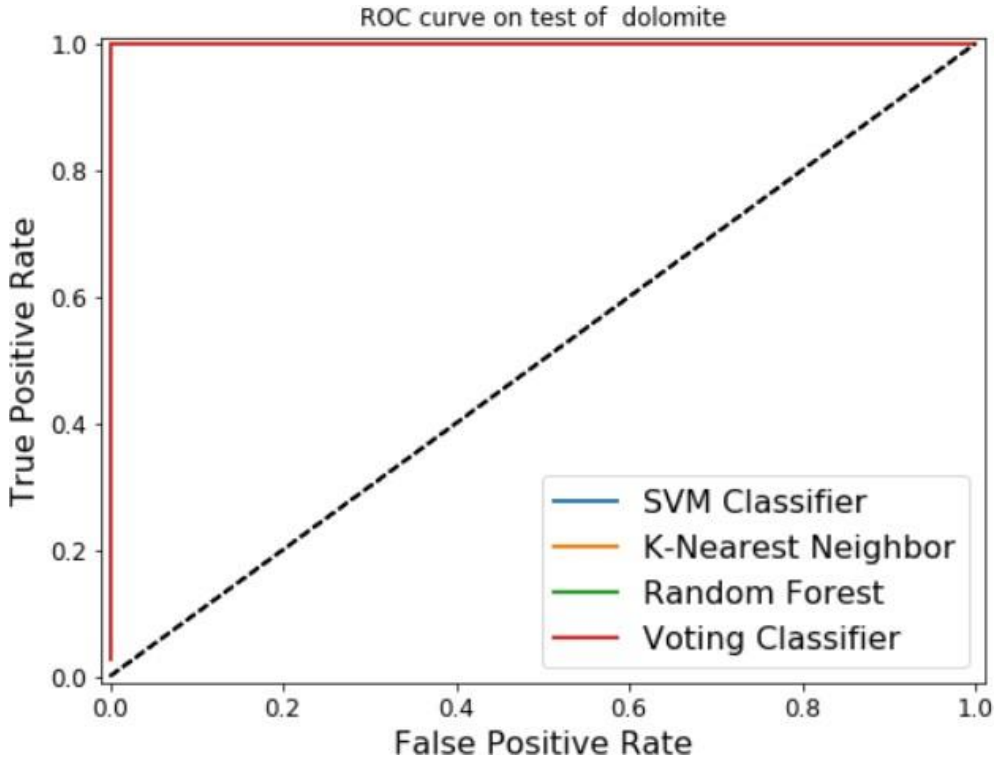

Figure 143: ROC curve of all classifiers on test-set of Dolomite

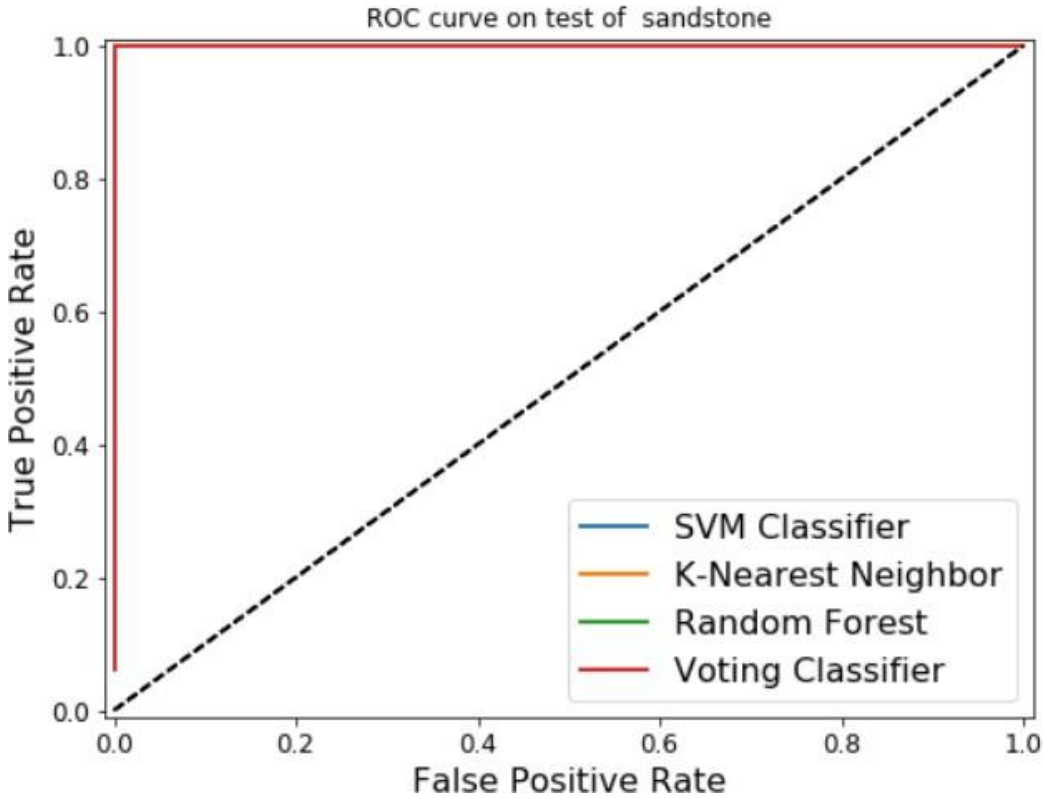

Figure 144:ROC curve of all classifiers on test-set of Sandstone

3.1.2.6.5.4 : Manual Voting

So far, we have been pretty much successful in getting almost perfect predictions on each class by the classifiers, but our aim is to increase the accuracies of the classifiers as much as possible. With that, we decided to do manual voting on the instances of the each of the classes in the test set. In manual voting(a function that we developed with help from Daniel (StackoverFlow, 2018 [5]), we improve the accuracies further by basically getting the confidence level ("*predict_proba*") of each classifier on the test set. Once we have the confidence levels, we can see on each instance of the class which of these classifiers is most confident and we would accept the answer from the most confident class. This will further improve the F1 & ROC-AUC scores as you will see. Note that this is also an example of the implementation of the Ensemble methods we learnt about earlier, this is why we decided it to call "*ensemble_test*" as we have combined three classifiers together as in an "*Ensemble method*".

We decided to test this first on the shaly limestone class. Once, we see improved scores we can simply automate the process as we did earlier for all other classifiers.

Manual Voting on Shaly Limestone:

In [284]:
```
# parameters for random forest
rfclf_params_sl = {
    'n_estimators': 500,
    'bootstrap': True,
    'class_weight':None,
    'criterion':'gini',
    'max_depth':None,
    'max_features':'auto',
    'random_state' : 41

    # ... fill in the rest you want here
}

# Fill in svm params here
svm_params_sl = {
    'C': 100,
    'probability':True,
    'random_state' : 42
}

# KNeighbors params go here
kneighbors_params_sl= {
    'n_neighbors': 5,
    'weights':'distance'
}

params_sl = [rfclf_params_sl, svm_params_sl, kneighbors_params_sl]
classifiers_sl = [RandomForestClassifier, SVC, KNeighborsClassifier]
```

Figure 145: Initiating parameters of Manual voting

```
def ensemble_test(classifiers, params, X_train, y_train, X_test):
    best_preds_test = np.zeros((len(X_test), 2))
    classes_test = np.unique(y_train)

    for i in range(len(classifiers)):
        # Construct the classifier by unpacking params
        # store classifier instance
        clf_test = classifiers[i](**params[i])
        # Fit the classifier as usual and call predict_proba
        clf_test.fit(X_train, y_train)
        y_preds_test = clf_test.predict_proba(X_test)
        # Take maximum probability for each class on each classifier
        # This is done for every instance in X_test
        best_preds_test = np.maximum(best_preds_test, y_preds_test)

    # map the maximum probability for each instance back to its corresponding class
    preds_test = np.array([classes_test[np.argmax(pred)] for pred in best_preds_test])
    return preds_test
```

Figure 146: Manual ensemble function [5]

The above code fits the classifiers with the parameters shown above one by one (i.e., parameters of SVM with its classifier, parameters of K-Nearest Neighbor with its classifier, etc., by using of unpacking). The purpose of implementing it this way is to make it easy for changing the classifiers parameters when necessary (for new data for example).

Each time the classifier then fits the data and return the confidence level of the classifiers. Next thing that the function does is find the maximum confidence level of the classifier on the instances and then finally return the classifier's output in terms of whether this class (for example shaly limestone) is actually true or not true instead of returning the confidence level only. Now we can check the performance as:

```
from sklearn.metrics import accuracy_score, f1_score
y_preds_test_sl = ensemble_test(classifiers_sl, params_sl, X_train, y_train_sl, X_test_prepared)
print('Accuracy score = ',accuracy_score(y_test_sl, y_preds_test_sl),'\n',
     'f1_score = ', f1_score(y_test_sl, y_preds_test_sl, average = 'weighted'),'\n',
     'roc_auc_score = ', roc_auc_score(y_test_sl, y_preds_test_sl, average = 'weighted'))
```

```
Accuracy score =  0.949514563107
 f1_score =  0.949653574035
 roc_auc_score =  0.933362369338
```

Figure 147: Accuracy scores of Manual voting on Shaly-Limestone class

If we go back and check the ROC-AUC scores of the shaly limestone class, we would see that this manual classifier performs even better than the original Voting classifier or Random Forest on the test set.

Now we can automate the process for all the classes:

```
# parameters for random forest
rfclf_params = {
    'n_estimators': 500,
    'bootstrap': True,
    'class_weight':None,
    'criterion':'gini',
    'max_depth':None,
    'max_features':'auto',
    'warm_start': True,
    'random_state': 41
    # ... fill in the rest you want here
}

# Fill in svm params here
svm_params = {
    'C': 100,
    'probability':True,
    'random_state':42
}

# KNeighbors params go here
kneighbors_params= {
    'n_neighbors': 5,
    'weights':'distance'
}
```

Figure 148: Code containing hyperparameters of all classifiers for generalization with all classes for Manual Voting

```
y_test_classes = (y_test_sl, y_test_lim, y_test_shale, y_test_sandlim, y_test_ss, y_test_dol, y_test_sand)
classifiers = [RandomForestClassifier, SVC, KNeighborsClassifier]
params = [rfclf_params, svm_params, kneighbors_params]
y_trains_classes= (y_train_sl, y_train_lim, y_train_shale, y_train_sandlim,
                   y_train_ss, y_train_dol, y_train_sand)
y_classes_names = ("shaly limestone", "limestone", "shale", "sandy lime",
                   "shaly sandstone", "dolomite", "sandstone")
```

Figure 149: Code containing all classes for generalization using Manual Voting

```
#Just get predictions
for y_trains, y_test, y_strings in zip(y_trains_classes, y_test_classes, y_classes_names):
    y_preds_test = ensemble_test(classifiers, params, X_train, y_trains, X_test_prepared)
    print("\n","Accuracy score for", y_strings, "=", accuracy_score(y_test, y_preds_test))
    print("f1_score for", y_strings, "=", f1_score(y_test, y_preds_test,
                                                   average = 'weighted', labels=np.unique(y_preds_test)))
    print("roc auc score for", y_strings, "=", roc_auc_score(y_test, y_preds_test,
                                                             average = 'weighted'))
```

Figure 150: Code to run the Manual voting for all classes

We get the following performance measures for all the classes below.

```
 Accuracy score for shaly limestone = 0.949514563107
f1_score for shaly limestone = 0.949653574035
roc auc score for shaly limestone = 0.933362369338

 Accuracy score for limestone = 0.957281553398
f1_score for limestone = 0.957272532095
roc auc score for limestone = 0.957311555515

 Accuracy score for shale = 0.95145631068
f1_score for shale = 0.948556595316
roc auc score for shale = 0.845505617978

 Accuracy score for sandy lime = 0.998058252427
f1_score for sandy lime = 0.998008114117
roc auc score for sandy lime = 0.95

 Accuracy score for shaly sandstone = 0.996116504854
f1_score for shaly sandstone = 0.998054474708
roc auc score for shaly sandstone = 0.5

 Accuracy score for dolomite = 1.0
f1_score for dolomite = 1.0
roc auc score for dolomite = 1.0

 Accuracy score for sandstone = 0.996116504854
f1_score for sandstone = 0.996226826208
roc auc score for sandstone = 0.997995991984
```

Figure 151: Accuracy measurements for all classes using Manual Voting classifier

As you can see that the classifiers don't perform well on shale in terms of ROC-AUC scores, although, there F-1 scores when compared to the automatic Voting Classifier and other classifiers are lower than our manual voting classifier. On the other hand, it out performs all other classifiers for all other classes of "*Types of Formations*".

The reason for ROC-AUC score on shaly sandstone is low simply because there are not enough training instances (called *under-fiting*) for the classifier to train and then make predictions on the test set. Similarly, for the shale class, there's a high probability that the ROC-AUC score is low because of underfitting again. Although, if we go back to the individual classifiers, we may find that SVM's ROC-AUC scores on both shale and shaly sandstone classes outruns all other classifiers including both the Voting classifiers.

A logical way of understanding this is that if Random Forest let's say performs well on one of the classes i.e., gives 85% accuracy like in shaly limestone class (850/1000 instances), while the SVM classifier is only 78% accurate (780/1000 times correct), now if the SVM classifier predict (780+150 =) 930 items the same as Random Forest, but gets the 70 instances wrong, it is easier to see how voting can do worse than Random Forest alone (SVM makes all the mistakes and then makes some more), so it's possible that one classifier regularized more strongly than combined for example. Thus, we can also expect SVM to be doing the same for the two classes below. More details about this can be found from the answers on Data Science Stack Exchange (2015) [6].

```
f1 score of actual prediction on test set of shale with SVMClassifier = 0.952510764485
roc auc score of actual prediction on test set of shale with SVMClassifier = 0.882744783307
```

```
f1 score of actual prediction on test set of shaly sandstone with SVMClassifier = 1.0
roc auc score of actual prediction on test set of shaly sandstone with SVMClassifier = 1.0
```

Figure 152: Improved performance on two of the classes using just the SVM classifier

For this purpose, if we had small dataset like ours with less instances to fit and predict, SVM would be the suitable choice to use. Again, both manual voting and automatic voting gives bad results as well because once combine it with the other two classifiers with bad predictions (more details of the effect discussed just above), the effect of SVM's predictions on the test set would suppress due to voting between

bad classifiers. Although, it would always be a better choice to test this on a new dataset with more instances of each formation to fit on.

Moreover, one last factor that might be important to consider during the voting between these classifiers is that because the data was actually pretty small for some for some of the formations (especially the shaly sandstone formation) we had to lower the cross-validation folds to “4” for all the classes training so that the algorithm doesn’t fail or give warnings due to lesser number of training or test instances to fit on (which were even less than the training instances). Again, if the data was sufficient we wouldn’t have to lower the cross-validation folds (cv) which thus would have increased the performance of both manual and automatic voting by sufficient training on the instances by the classifiers.

4.1: Conclusion & Recommendation

With Python's Machine Learning algorithms, we successfully trained a dataset that contains geological log information. Here's a summary of what we did.

When dealing with geological log data like this, usually we would require to visualize the data before cleaning and see what part of the data might decrease the performance of the ML system. The data with off points are removed followed by taking care of missing values in the data (either by removing them or by using "*imputer*".

We can then visualize the data after cleaning and then deal with text-classification problem as ours which the ML algorithms cannot handle without conversion to either Boolean form (*OneHotEncoder* or *Get_Dummies*) as we did or converting them into integers values (by *LabelBinazer*). The data is then ready to be trained on different ML algorithms (present built-in Python's Scikit-Learn's libraries). We then check accuracies of the predictions by different methods as discussed and visualize the performance on our dataset as shown (ROC-curves for Boolean classification).

We also saw some important correlation between the logs with each other through a *correlation matrix* and the relation of different "*types of formation*" with the logs which suggest the relative importance of each log with different rock type through a *feature importance.* If both of these are used correctly with more data of such type and practically tested in the formation evaluation labs and further field practice, we can investing millions in logs which do not have relative importance in prediction of the type of formation or formation fluids.

Despite our success to some extent of training the system using ML algorithm on our geological log dataset, the biggest challenge was to make the algorithms work on the limited data that we had, because of which we saw reduction of the performance of the algorithms on the test sets.

Even with this limitation, we were able to make the algorithms work around the problem and make predictions about the type of formation with almost 93-99% accuracy which could have gone to almost a 99.9% perfect if we had more data. In the future, it would be ideal to start with a new dataset with more instances of each type of formation and adopt a similar strategy as highlighted to train the data by ML algorithms and improve accuracy of the predictions further.

Also, we would like to point out that we didn't try to implement the algorithm using the text targets ("*Types of formation*") as integer, which is another way to work around with this type of text-classification problem. Due to limitation of time we couldn't approach the problem this way which is also a promising method of multiclass-classification problem like this.

We mentioned earlier as well that the data that was used for this classification problem was from a Kanas Geological Survey's public domain [7], from which similar any kind of data can be downloaded for training the ML system (irrespective of the type of formations we dealt with in our training example) with the exception of interpreting the "Types of formations" before implementing this *supervised machine learning* technique. Our algorithm would work for all types of formations in the dataset if trained correctly and predict the type of formation according to the way system was trained with presence of sufficient data.

5.1: References

1- GÃ©ron, Aurelien. *Hands-on Machine Learning with Scikit-Learn and TensorFlow: Concepts, Tools, and Techniques to Build Intelligent Systems*. O'Reilly Media, 2017.

2- “Scikit-Learn.” *Scikit-Learn: Machine Learning in Python - Scikit-Learn 0.19.1 Documentation*, scikit-learn.org/stable/.

3- “Classification with Scikit-Learn.” *Ahmed Taspinar*, 1 Mar. 2018, www.ataspinar.com/2017/05/26/classification-with-scikit-learn/.

4- Ray, Sunil, et al. “6 Easy Steps to Learn Naive Bayes Algorithm (with Code in Python).” *Analytics Vidhya*, 11 Apr. 2018, www.analyticsvidhya.com/blog/2017/09/naive-bayes-explained/.

5- “Improving the Prediction Score by Use of Confidence Level of Classifiers on Instances.” *Python - Improving the Prediction Score by Use of Confidence Level of Classifiers on Instances - Stack Overflow*, www.stackoverflow.com/questions/49396961/improving-the-prediction-score-by-use-of-confidence-level-of-classifiers-on-inst.

6- “Voting Combined Results from Different Classifiers Gave Bad Accuracy.” *Machine Learning - Voting Combined Results from Different Classifiers Gave Bad Accuracy - Data Science Stack Exchange*, www.datascience.stackexchange.com/questions/8339/voting-combined-results-from-different-classifiers-gave-bad-accuracy.

7- *KGS--Oil and Gas Well Database*, www.kgs.ku.edu/Magellan/Qualified/index.html.