Data Understanding and Cleaning

Let's understand the dataset and see if it needs some cleaning etc.

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn import linear_model
from sklearn.model_selection import train_test_split
import gc
import cv2
```

```
# read the dataset
digits = pd.read_csv("digit_svm.csv")
digits.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 42000 entries, 0 to 41999
Columns: 785 entries, label to pixel783
dtypes: int64(785)
memory usage: 251.5 MB
```

```
# head
digits.head()
```

	label	pixel0	pixel1	pixel2	pixel3	pixel4	pixe
0	1	0	0	0	0	0	
1	0	0	0	0	0	0	

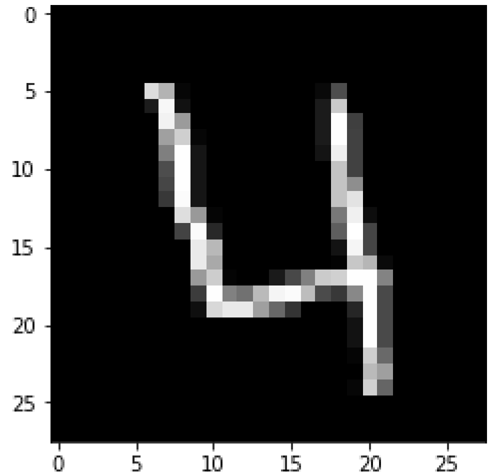
```
four = digits.iloc[3, 1:]
four.shape
```

```
(784,)
```

5 rows × 785 columns

```
four = four.values.reshape(28, 28)
plt.imshow(four, cmap='gray')
```

```
<matplotlib.image.AxesImage at 0x1a26b33208>
```



Side note: Indexing Recall

```
list = [0, 4, 2, 10, 22, 101, 10]
indices = [0, 1, 2, 3, ..., ]
reverse = [-n -3 -2 -1]
```

```
# visualise the array
print(four[5:-5, 5:-5])
```

```
[[  0 220 179   6   0   0   0   0   0   0   0   0   9
 [  0  28 247  17   0   0   0   0   0   0   0   0  27
 [  0   0 242 155   0   0   0   0   0   0   0   0  27
 [  0   0 160 207   6   0   0   0   0   0   0   0  27
 [  0   0 127 254  21   0   0   0   0   0   0   0  20
 [  0   0  77 254  21   0   0   0   0   0   0   0   0
 [  0   0  70 254  21   0   0   0   0   0   0   0   0
 [  0   0  56 251  21   0   0   0   0   0   0   0   0
 [  0   0   0 222 153   5   0   0   0   0   0   0   0
 [  0   0   0  67 251  40   0   0   0   0   0   0   0
 [  0   0   0   0 234 184   0   0   0   0   0   0   0
 [  0   0   0   0 234 169   0   0   0   0   0   0   0
 [  0   0   0   0 154 205   4   0   0  26  72 128 203
 [  0   0   0   0  61 254 129 113 186 245 251 189  75
 [  0   0   0   0  15 216 233 233 159 104  52   0   0
 [  0   0   0   0   0   0   0   0   0   0   0   0   0
 [  0   0   0   0   0   0   0   0   0   0   0   0   0
 [  0   0   0   0   0   0   0   0   0   0   0   0   0
```

```
# Summarise the counts of 'label' to see how many labels of
digits.label.value_counts()
```

```
1    4684
7    4401
3    4351
9    4188
2    4177
6    4137
0    4132
4    4072
8    4063
5    3795
Name: label, dtype: int64
```

```
# Summarise count in terms of percentage
```

```
100*(round(digits.label.astype('category').value_counts()/le
```

```
1    11.15
7    10.48
3    10.36
9     9.97
2     9.95
6     9.85
0     9.84
4     9.70
8     9.67
5     9.04
Name: label, dtype: float64
```

Thus, each digit/label has an approximately 9%-11% fraction in the dataset and the **dataset is balanced**. This is an important factor in considering the choices of models to be used, especially SVM, since **SVMs rarely perform well on imbalanced data** (think about why that might be the case).

Let's quickly look at missing values, if any.

```
# missing values - there are none
digits.isnull().sum()
```

```
pixel2      0
pixel3      0
pixel4      0
pixel5      0
pixel6      0
pixel7      0
pixel8      0
pixel9      0
pixel10     0

pixel11     0
pixel12     0
pixel13     0
```

```
pixel14     0
pixel15     0
pixel16     0
pixel17     0
pixel18     0
pixel19     0
pixel20     0
pixel21     0
pixel22     0
pixel23     0
pixel24     0
pixel25     0
pixel26     0
pixel27     0
pixel28     0
           ..
pixel754    0
pixel755    0
pixel756    0
pixel757    0
pixel758    0
pixel759    0
pixel760    0
pixel761    0
pixel762    0
pixel763    0
pixel764    0
pixel765    0
pixel766    0
pixel767    0
pixel768    0
pixel769    0
pixel770    0
pixel771    0
pixel772    0
pixel773    0
pixel774    0
pixel775    0
pixel776    0
pixel777    0
pixel778    0
pixel779    0
```

```
pixel780    0
pixel781    0
pixel782    0
pixel783    0
Length: 785, dtype: int64
```

Also, let's look at the average values of each column, since we'll need to do some rescaling in case the ranges vary too much.

```
# average values/distributions of features
description = digits.describe()
description
```

	label	pixel0	pixel1	pixel2	pixel3
count	42000.000000	42000.0	42000.0	42000.0	42000.0
mean	4.456643	0.0	0.0	0.0	0.0
std	2.887730	0.0	0.0	0.0	0.0
min	0.000000	0.0	0.0	0.0	0.0
25%	2.000000	0.0	0.0	0.0	0.0
50%	4.000000	0.0	0.0	0.0	0.0
75%	7.000000	0.0	0.0	0.0	0.0
max	9.000000	0.0	0.0	0.0	0.0

8 rows × 785 columns

You can see that the max value of the mean and maximum values of some features (pixels) is 139, 255 etc., whereas most features lie in much lower ranges (look at description of pixel 0, pixel 1 etc. above).

Thus, it seems like a good idea to rescale the features.

Data Preparation for Model Building

Let's now prepare the dataset for building the model. We'll only use a fraction of the data else training will take a long time.

```
# Creating training and test sets
# Splitting the data into train and test
X = digits.iloc[:, 1:]
Y = digits.iloc[:, 0]

# Rescaling the features
from sklearn.preprocessing import scale
X = scale(X)

# train test split with train_size=10% and test size=90%
x_train, x_test, y_train, y_test = train_test_split(X, Y, tr
print(x_train.shape)
print(x_test.shape)
print(y_train.shape)
print(y_test.shape)
```

```
/anaconda3/lib/python3.6/site-packages/sklearn/model_s
  FutureWarning)
(4200, 784)
(37800, 784)
(4200,)
(37800,)
```

```
# delete test set from memory, to avoid a memory error
# we'll anyway use CV to evaluate the model, and can use the
```

```
# to evaluate the model finally

# del x_test
# del y_test
```

Model Building

Let's now build the model and tune the hyperparameters. Let's start with a **linear model** first.

Linear SVM

Let's first try building a linear SVM model (i.e. a linear kernel).

```
from sklearn import svm
from sklearn import metrics

# an initial SVM model with linear kernel
svm_linear = svm.SVC(kernel='linear')

# fit
svm_linear.fit(x_train, y_train)
```

```
SVC(C=1.0, cache_size=200, class_weight=None, coef0=0.
  decision_function_shape='ovr', degree=3, gamma='auto
  max_iter=-1, probability=False, random_state=None, s
  tol=0.001, verbose=False)
```

```
# predict
predictions = svm_linear.predict(x_test)
predictions[:10]
```

```
array([1, 3, 0, 0, 1, 9, 1, 5, 0, 6])
```

```
# evaluation: accuracy
# C(i, j) represents the number of points known to be in cla
# but predicted to be in class j
confusion = metrics.confusion_matrix(y_true = y_test, y_pred
confusion
```

```
array([[3615,    0,   12,    8,    8,   28,   28,    5
       [   0, 4089,   16,   23,    9,    3,    3,   13
       [  54,   48, 3363,   64,   74,   13,   53,   52
       [  20,   28,  121, 3387,    8,  175,    5,   54
       [  12,   12,   26,    2, 3399,    7,   41,   41
       [  49,   42,   32,  177,   41, 2899,   54,   14
       [  36,   16,   55,    5,   34,   37, 3486,    3
       [   9,   27,   37,   22,   70,   10,    4, 3619
       [  26,   86,   71,  137,   24,  137,   29,   26
       [  38,   11,   39,   26,  182,   19,    1,  207
```

```
# measure accuracy
metrics.accuracy_score(y_true=y_test, y_pred=predictions)
```

```
0.9042592592592592
```

```
# class-wise accuracy
class_wise = metrics.classification_report(y_true=y_test, y_
print(class_wise)
```

```
              precision    recall  f1-score   support

           0       0.94      0.97      0.95      3715
           1       0.94      0.98      0.96      4185
           2       0.89      0.89      0.89      3790
           3       0.88      0.87      0.87      3900
           4       0.88      0.92      0.90      3702
           5       0.87      0.85      0.86      3418
           6       0.94      0.94      0.94      3693
           7       0.90      0.92      0.91      3954
```

```
              8       0.91      0.84      0.88      3665
              9       0.88      0.85      0.87      3778

    avg / total       0.90      0.90      0.90     37800
```

```
# run gc.collect() (garbage collect) to free up memory
# else, since the dataset is large and SVM is computationall
# it'll throw a memory error while training
gc.collect()
```

```
87
```

Non-Linear SVM

Let's now try a non-linear model with the RBF kernel.

```
# rbf kernel with other hyperparameters kept to default
svm_rbf = svm.SVC(kernel='rbf')
svm_rbf.fit(x_train, y_train)
```

```
SVC(C=1.0, cache_size=200, class_weight=None, coef0=0.
  decision_function_shape='ovr', degree=3, gamma='auto
  max_iter=-1, probability=False, random_state=None, s
  tol=0.001, verbose=False)
```

```
# predict
predictions = svm_rbf.predict(x_test)
```

```
# accuracy
print(metrics.accuracy_score(y_true=y_test, y_pred=predictio
```

```
0.925582010582
```

The accuracy achieved with a non-linear kernel is slightly higher than a linear one. Let's now do a grid search CV to tune the hyperparameters C and gamma.

Grid Search Cross-Validation

```
# conduct (grid search) cross-validation to find the optimal
# of cost C and the choice of kernel

from sklearn.model_selection import GridSearchCV

parameters = {'C':[1, 10, 100],
             'gamma': [1e-2, 1e-3, 1e-4]}

# instantiate a model
svc_grid_search = svm.SVC(kernel="rbf")

# create a classifier to perform grid search
clf = GridSearchCV(svc_grid_search, param_grid=parameters, s

# fit
clf.fit(x_train, y_train)
```

```
GridSearchCV(cv=None, error_score='raise',
       estimator=SVC(C=1.0, cache_size=200, class_weig
  decision_function_shape='ovr', degree=3, gamma='auto
  max_iter=-1, probability=False, random_state=None, s
  tol=0.001, verbose=False),
       fit_params=None, iid=True, n_jobs=1,
       param_grid={'C': [1, 10, 100], 'gamma': [0.01,
       pre_dispatch='2*n_jobs', refit=True, return_tra
       scoring='accuracy', verbose=0)
```

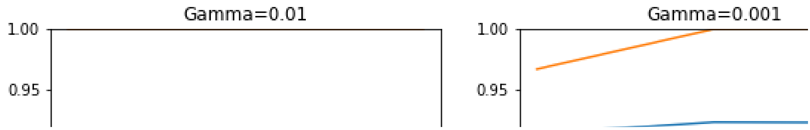
```
# results
```

```
cv_results = pd.DataFrame(clf.cv_results_)
cv_results
```

```
/Library/Frameworks/Python.framework/Versions/3.5/lib/
  warnings.warn(*warn_args, **warn_kwargs)
/Library/Frameworks/Python.framework/Versions/3.5/lib/
  warnings.warn(*warn_args, **warn_kwargs)
/Library/Frameworks/Python.framework/Versions/3.5/lib/
  warnings.warn(*warn_args, **warn_kwargs)
/Library/Frameworks/Python.framework/Versions/3.5/lib/
  warnings.warn(*warn_args, **warn_kwargs)
/Library/Frameworks/Python.framework/Versions/3.5/lib/
  warnings.warn(*warn_args, **warn_kwargs)
```

```
# converting C to numeric type for plotting on x-axis
cv_results['param_C'] = cv_results['param_C'].astype('int')

# # plotting
plt.figure(figsize=(16,6))

# subplot 1/3
plt.subplot(131)
gamma_01 = cv_results[cv_results['param_gamma']==0.01]

plt.plot(gamma_01["param_C"], gamma_01["mean_test_score"])
plt.plot(gamma_01["param_C"], gamma_01["mean_train_score"])
plt.xlabel('C')
plt.ylabel('Accuracy')
plt.title("Gamma=0.01")
plt.ylim([0.60, 1])
plt.legend(['test accuracy', 'train accuracy'], loc='lower r
plt.xscale('log')

# subplot 2/3
plt.subplot(132)
gamma_001 = cv_results[cv_results['param_gamma']==0.001]

plt.plot(gamma_001["param_C"], gamma_001["mean_test_score"])
plt.plot(gamma_001["param_C"], gamma_001["mean_train_score"]
plt.xlabel('C')
plt.ylabel('Accuracy')
plt.title("Gamma=0.001")
```

```
plt.ylim([0.60, 1])
plt.legend(['test accuracy', 'train accuracy'], loc='lower r
plt.xscale('log')


# subplot 3/3
plt.subplot(133)
gamma_0001 = cv_results[cv_results['param_gamma']==0.0001]

plt.plot(gamma_0001["param_C"], gamma_0001["mean_test_score"
plt.plot(gamma_0001["param_C"], gamma_0001["mean_train_score
plt.xlabel('C')
plt.ylabel('Accuracy')
plt.title("Gamma=0.0001")
plt.ylim([0.60, 1])
plt.legend(['test accuracy', 'train accuracy'], loc='lower r
plt.xscale('log')

plt.show()
```



From the plot above, we can observe that (from higher to lower gamma / left to right):

- At very high gamma (0.01), the model is achieving 100% accuracy on the training data, though the test score is quite low (<75%). Thus, the model is overfitting.
- At gamma=0.001, the training and test scores are comparable at around C=1, though the model starts to overfit at higher values of C
- At gamma=0.0001, the model does not overfit till C=10 but starts showing signs at C=100. Also, the training and test scores are slightly lower than at gamma=0.001.

Thus, it seems that the best combination is gamma=0.001 and C=1 (the plot in the middle), which gives the highest test accuracy (~92%) while avoiding overfitting.

Let's now build the final model and see the performance on test data.

Final Model

Let's now build the final model with chosen hyperparameters.

```
# optimal hyperparameters
best_C = 1
best_gamma = 0.001

# model
```

```
# model
svm_final = svm.SVC(kernel='rbf', C=best_C, gamma=best_gamma

# fit
svm_final.fit(x_train, y_train)
```

```
SVC(C=1, cache_size=200, class_weight=None, coef0=0.0,
  decision_function_shape='ovr', degree=3, gamma=0.001
  max_iter=-1, probability=False, random_state=None, s
  tol=0.001, verbose=False)
```

```
# predict
predictions = svm_final.predict(x_test)


# evaluation: CM
confusion = metrics.confusion_matrix(y_true = y_test, y_pred

# measure accuracy
test_accuracy = metrics.accuracy_score(y_true=y_test, y_pred

print(test_accuracy, "\n")
print(confusion)
```

```
0.924973544974

[[3587    0   10   10    5   15   50   12   25    1]
 [   0 4108   14   16    5    3    6   18   10    5]
 [  24   23 3407   65   44    5   36  123   54    9]
 [   4   21   86 3502    5   89   11   73   76   33]
 [   3   11   36    7 3450   13   23   43    6  110]
 [  20   29   14  114   18 3020   79   53   36   35]
 [  31   12   11    1   14   34 3521   44   25    0]
 [   4   28   27    8   36    7    1 3739    7   97]
 [  14   59   32   80   22   97   25   44 3251   41]
 [  23   13   13   50   98    7    0  176   19 3379]]
```

Conclusion

The final accuracy on test data is approx. 92%. Note that this can be significantly increased by using the entire training data of 42,000 images (we have used just 10% of that!).