ECE 532 Final Project

The MNIST is a “Hello World” project in machine learning (ML). At the same time, it is one of the most appropriate databases for machine learning beginners who want to practice their machine learning techniques. This MNIST dataset contains real-world hand-written images which makes it become popular for machine learning projects. During these years, more and more machine learning algorithms researchers improve this MNIST project in order to help ML beginners. In this project, we study several ML solvers using **sk.learn** and applied those on the MNIST dataset. These ML solvers include: **linear ridge regression**, **nearest neighbor classification** and **kernel based support vector machine(SVM)**.

1 Downloading the dataset

1.1 Downloading the MNIST

The MNIST is a large database of handwritten digits. There are 60,000 images of 28x28 pixel in the training set, and the testing set is composed of 10,000 patterns of 28x28 pixel. Each pixel forms a feature, so one image has 784 features, and the value range of each pixel is [0, 255]. The training label set has 60,000 numbers, and the testing label set contains 10,000 results. Each label is an actual number between 0-9. We can use *mnist = fetch_openml("mnist_784")* to download the MNIST.

```
#1 Downloading the Data (MNIST)
import numpy as np
from sklearn.datasets import fetch_openml
mnist = fetch_openml("mnist_784")
```

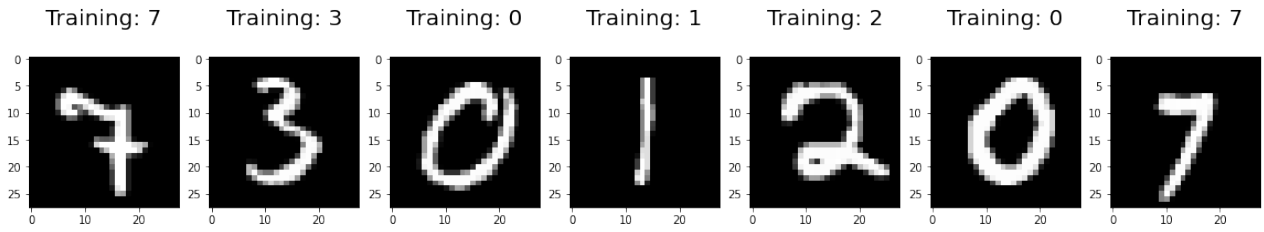
1.2 Splitting the data into Training and Test Sets

Using *train_test_split* separates the MNIST into the traing subset and the test subset. The training set has 60,000 images and corresponding labels. The test set involves 10,000 images and corresponding labels.

```
#2 Splitting the MNIST
from sklearn.model_selection import train_test_split
train_img, test_img, train_lbl, test_lbl = train_test_split(mnist.data,
mnist.target, test_size=1/7.0, random_state=0)
train_lbl = np.asarray(train_lbl,'float64')
test_lbl = np.asarray(test_lbl,'float64')
train_img.shape
test_img.shape
```

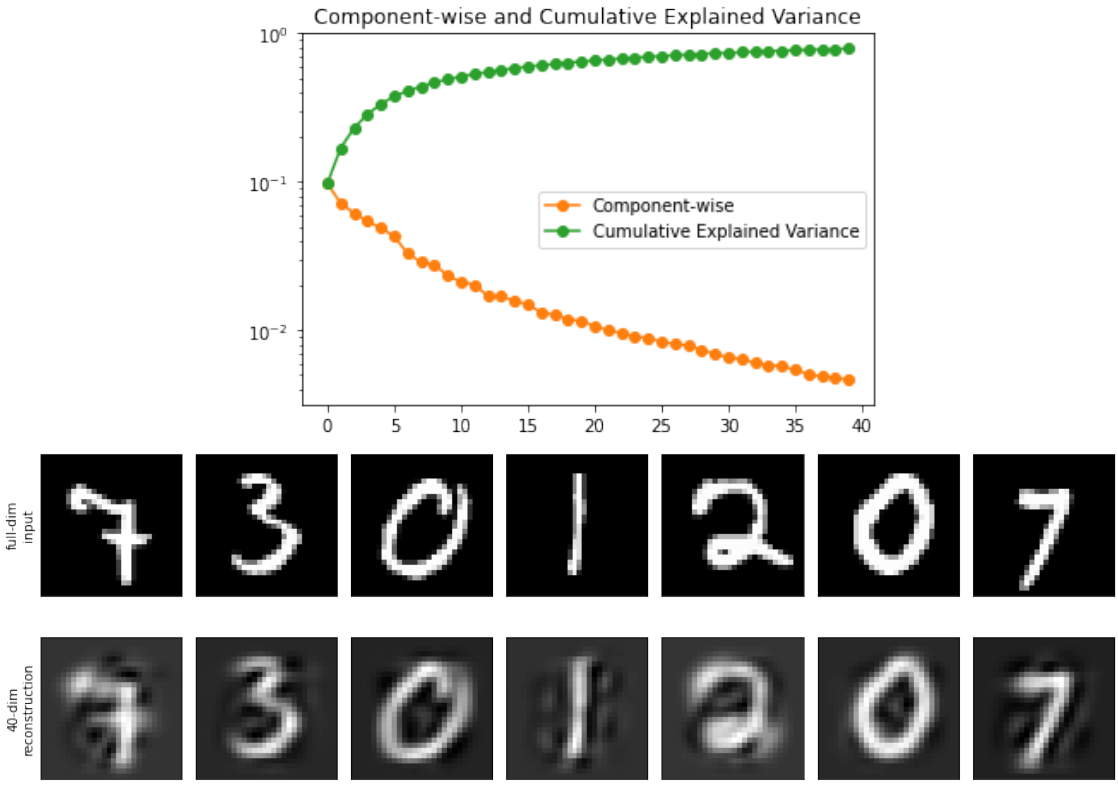
1.3 Training data display

```
# Data display
import matplotlib.pyplot as plt
plt.figure(figsize=(20,4))
for index, (image, label) in enumerate(zip(train_img[0:7], train_lbl[0:7])):
    plt.subplot(1, 7, index + 1)
    plt.imshow(np.reshape(image, (28,28)), cmap=plt.cm.gray)
    plt.title('Training: %i\n' % label, fontsize = 20)
```



2 Dimensionality reduction

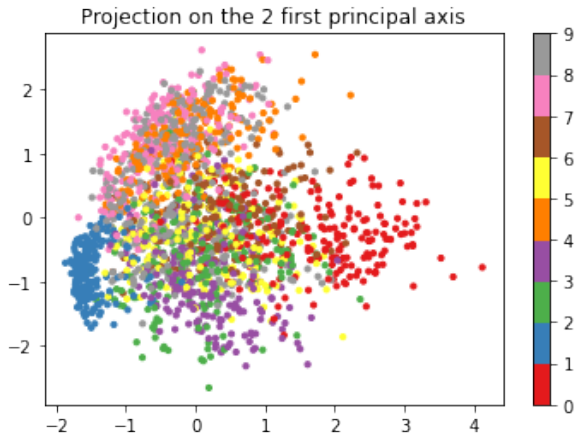
The MNIST has a huge computation in a learning process, so we need to use principal component analysis (PCA) to reduce its dimensionality. For PCA, the higher cumulative which explained variance of principal components is chosen, the more information in the original features can be retained. we choose the first 40 components as the new features that collects 78.7% of variance. Also, we can directly find the difference between the original image and the reconstruction image using the first 40 components.




```
# PCA with the first 40 components
from sklearn.decomposition import PCA
```

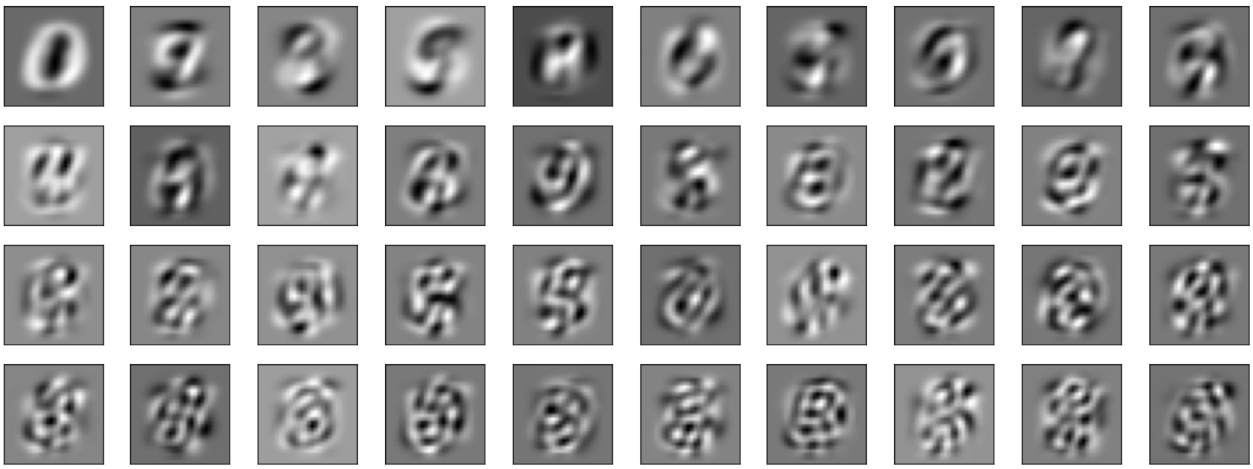
```
from time import time
n_components = 40
t0 = time()
pca = PCA(n_components=n_components, svd_solver='randomized',
          whiten=True).fit(train_img)
print("done in %0.3fs" % (time() - t0))
train_img_pca = pca.transform(train_img)

# Component-wise and Cumulative Explained Variance
plt.hist(pca.explained_variance_ratio_, bins=n_components, log=True)
sum_var = pca.explained_variance_ratio_.sum()
plt.plot(range(40), pca.explained_variance_ratio_, 'o-', label='Component-wise')
plt.plot(range(40), np.cumsum(pca.explained_variance_ratio_), 'o-', label='Cumulative Explained Variance')
plt.title("Component-wise and Cumulative Explained Variance")
plt.legend()
plt.ylim(0, 1)
print(sum_var)

# Reconstruct the original image using the new features.
projected_pca = pca.inverse_transform(train_img_pca)
fig, ax = plt.subplots(2, 7, figsize=(15, 5),
                       subplot_kw={'xticks':[], 'yticks':[]},
                       gridspec_kw=dict(hspace=0.1, wspace=0.1))
for i in range(7):
    ax[0, i].imshow(train_img[i].reshape(28, 28), cmap=plt.cm.gray)
    ax[1, i].imshow(projected_pca[i].reshape(28, 28), cmap=plt.cm.gray)

ax[0, 0].set_ylabel('full-dim\ninput')
ax[1, 0].set_ylabel('40-dim\nreconstruction');
```

The visualization of data using PCA that shows that the multi-class classification is in the 2-dimensions.



```
# Visualization of the MNIST dataset in 2D
plt.scatter(train_img_pca[0:2000, 0], train_img_pca[0:2000, 1],
c=train_lbl[0:2000], s=10, cmap='Set1')
plt.colorbar()
plt.title('Projection on the 2 first principal axis')
```

The components are ordered by their importance from top-left to bottom-right in the following figure. We see that the first few components seem to depict the outline, and the remaining components highlight more details.



```
# The ordered components display
fig = plt.figure(figsize=(16, 6))
for i in range(40):
    ax = fig.add_subplot(4, 10, i + 1, xticks=[], yticks=[])
    ax.imshow(pca.components_[i].reshape(np.reshape(train_img[0],
(28,28)).shape), cmap=plt.cm.gray)
```

3 Three algorithms and results

3.1 Linear ridge regression classifier

Ridge regression imposing a penalty on the size of the coefficients based on the ordinary Least Squares problem to control the amount of shrinkage. The ridge coefficients minimize a penalized residual sum of squares. Its cost function can be expressed as

$$\min_{w} ||Xw - y||_2^2 + \alpha||w||_2^2, \quad \alpha \geq 0$$

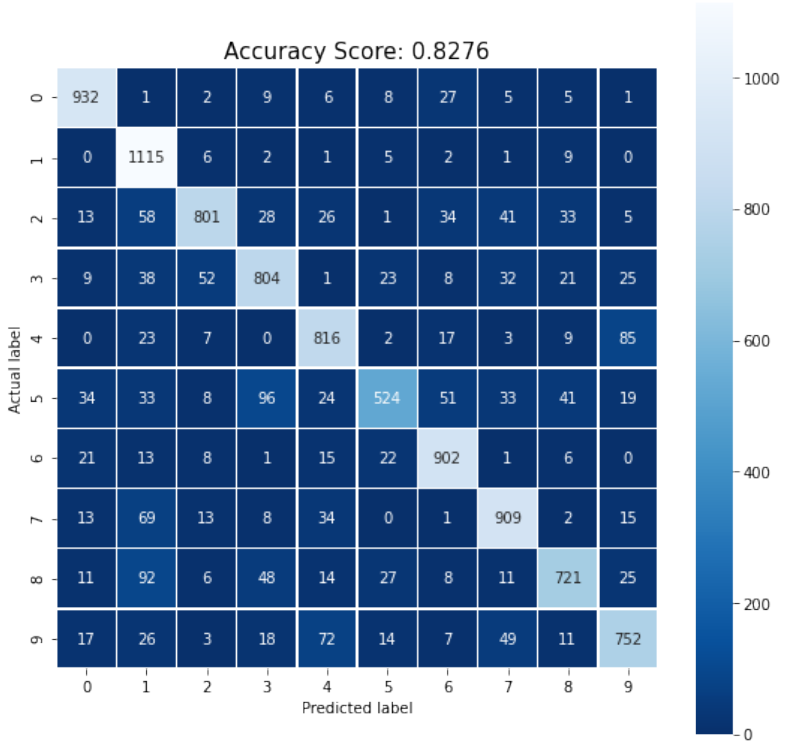
The first term is the loss function of least squares, and the second term is a penalty term for the ridge regression. The key parameter α is a trade off between variance and bias. Thus, it is necessary to use cross-validation to choose an appropriate α. The Ridge regressor has a classifier variant: *RidgeClassifier*, which can achieve the multiclass classification. Besides, we need to use cross-validation to determine the best α. We set the range of α from 10^{-7} to 10^{6}. Finally, we find that indicates Various α have the similar performance except for α=10^{6}.

Table 1: Performance summary of different α

α	10^{-7}	10^{-5}	10^{-3}	10^{-1}	10^{2}	10^{4}	10^{6}
Score	**0.8346**	0.8346	0.8346	0.8346	0.8346	0.8341	0.7447

```
# Learning using the best parameters and prediction_ridge regression
from sklearn.linear_model import RidgeClassifier
from sklearn.model_selection import GridSearchCV
param_grid = [
  {'alpha':[1e-7, 4e-7, 8e-7, 1e-6, 4e-6, 8e-6, 1e-5, 4e-5, 8e-5, 1e-4, 4e-4, 8e-4, 1e-4, 1e-2, 1e-1, 1e2, 1e4, 1e6]}
  ]
t0 = time()
reg_clsf = RidgeClassifier()
reg_grid_clsf = GridSearchCV(reg_clsf,param_grid,n_jobs=1, verbose=2, cv=10)
reg_grid_clsf.fit(train_img_pca, train_lbl)
print("done in %0.3fs" % (time() - t0))
reg_classifier = reg_grid_clsf.best_estimator_
reg_params = reg_grid_clsf.best_params_
# results of cross-validation
pd.pivot_table(pd.DataFrame(reg_grid_clsf.cv_results_),
               values='mean_test_score', index='param_alpha')

# prediction
reg_predictions = reg_classifier.predict(pca.transform(test_img))
# evalution
reg_score = reg_classifier.score(pca.transform(test_img), test_lbl)
print(reg_score)
```

Using the best α of **10^{-7}** fit the model on the entire training data(60000), and predict new labels on the entire testing data, then score this method. The final score is **0.8276**.




```
# Confusion matrix
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn import metrics
cm = metrics.confusion_matrix(test_lbl, reg_predictions)
print(cm)

plt.figure(figsize=(9,9))
sns.heatmap(cm, annot=True, fmt="1d", linewidths=.5, squsre = True, cmap =
'Blues_r');
plt.ylabel('Actual label');
plt.xlabel('Predicted label');
all_sample_title = 'Accuracy Score: {0}'.format(reg_score)
plt.title(all_sample_title, size = 15);
```

3.2 KNN classifier

Nearest neighbor classification is a non-linear instance-based learning classifier. It simply stores instances from the training data and compare distances between unknown data with the training data. Then based on distances, it predicts labels for the unknown data.

KNN requires a total of X training data points and M classes, it predicts an unobserved training point X~new~ as the mean of the closes k neighbours to X~new~. Usually, the standard Euclidean metric $d(x, x_i) = \sqrt{\sum_{i=1}^{n}(x - x_i)^2}$ is used as the the distance matrix.

$\hat{y} = \frac{1}{k}\sum_{x_i \in X} d(x_i, x)$

There is a nearest neighbor classifier named *KNeighborsClassifier* that implements learning based on a k nearest neighbors of the new data. When k is too small, the model becomes susceptible to noise and outlier data points; however, if k is too large, it poses a risk over-smoothing the classification results and increasing bias. It need to learn the training data using different values k, then chooses the best according to their performance of cross-validation.

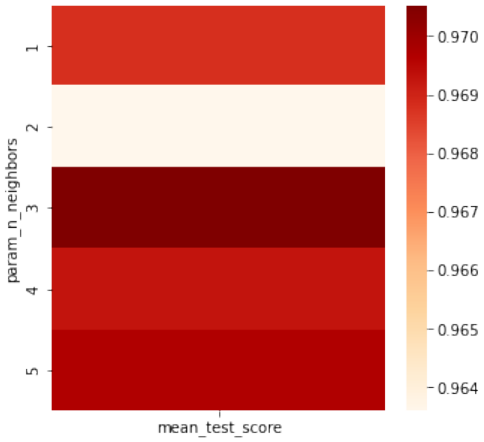


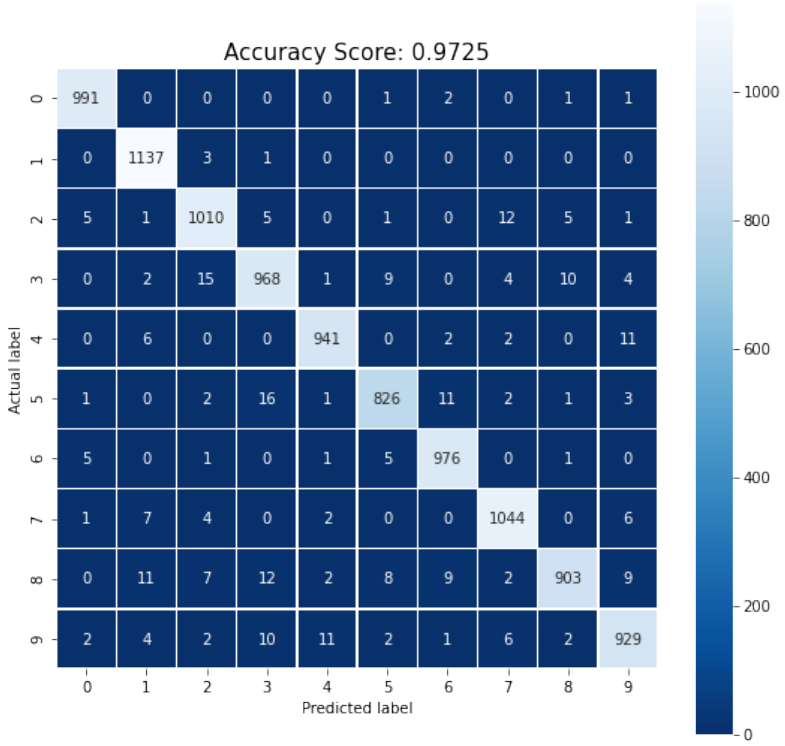
Table 2: Performance summary of different k

k	1	2	**3**	4	5
Score	0.9688	0.9636	**0.9725**	0.9693	0.9697

```
# Learning using the best parameters and prediction_KNN
from sklearn import neighbors
param_grid = [
  {'n_neighbors':[1, 2, 3, 4, 5]}
  ]
t0 = time()
knn_clsf = neighbors.KNeighborsClassifier()
knn_grid_clsf = GridSearchCV(knn_clsf,param_grid,n_jobs=1, verbose=2,
cv=10)
knn_grid_clsf.fit(train_img_pca, train_lbl)
print("done in %0.3fs" % (time() - t0))
knn_classifier = knn_grid_clsf.best_estimator_
knn_params = knn_grid_clsf.best_params_
# results of cross-validation
pd.pivot_table(pd.DataFrame(knn_grid_clsf.cv_results_),
values='mean_test_score', index='param_n_neighbors')

# heatmap to display the results of cross-validation.
import seaborn as sns
import pandas as pd
pvt = pd.pivot_table(pd.DataFrame(knn_grid_clsf.cv_results_),
values='mean_test_score', index='param_n_neighbors')
plt.figure(figsize=(5, 5))
sns.heatmap(pvt,cmap = 'OrRd')
```

```
22
23  # prediction
24  knn_predictions = knn_classifier.predict(pca.transform(test_img))
25  # evalution
26  knn_score = knn_classifier.score(pca.transform(test_img), test_lbl)
27  print(knn_score)
```

Using the best k of **3** fit the model on the entire training data(60000), and predict new labels on the entire testing data. The final score is **0.9725**.



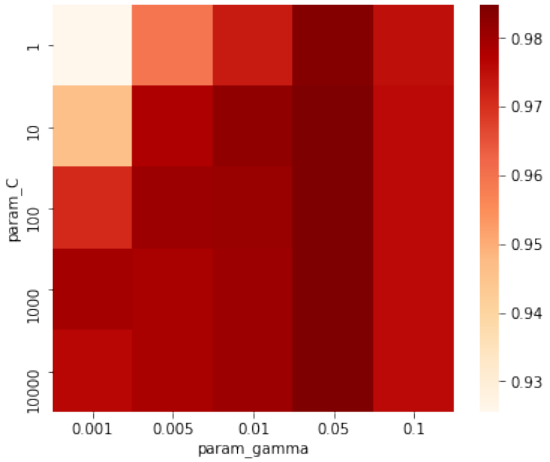

```
1   # confusion matrix
2   import matplotlib.pyplot as plt
3   import seaborn as sns
4   from sklearn import metrics
5   cm = metrics.confusion_matrix(test_lbl, knn_predictions)
6   print(cm)
7
8   plt.figure(figsize=(9,9))
9   sns.heatmap(cm, annot=True, fmt="1d", linewidths=.5, square = True, cmap =
    'Blues_r');
10  plt.ylabel('Actual label');
11  plt.xlabel('Predicted label');
12  all_sample_title = 'Accuracy Score: {0}'.format(knn_score)
13  plt.title(all_sample_title, size = 15);
```

3.3 Kernel based SVM

Support Vector Machine is capable of performing multi-class classification on a large dataset, which maximizes the margin around the separating hyperplane and only depends on the support vectors. Kernel based support vector machines can be written as following. These kernel functions has various forms, such as polynomial $K(u,v) = (u^T v + 1)^q$ or Gaussian $K(u,v) = e^{-\frac{||u-v||_2^2}{2\sigma^2}}$.

$$\min_{\alpha}(1 - y^i \sum_{j=1}^{N} \alpha_j K(x^i x^j))_+ + \lambda \sum_{i=1}^{N} \sum_{j=1}^{N} \alpha_i \alpha_j K(x^i x^j)$$

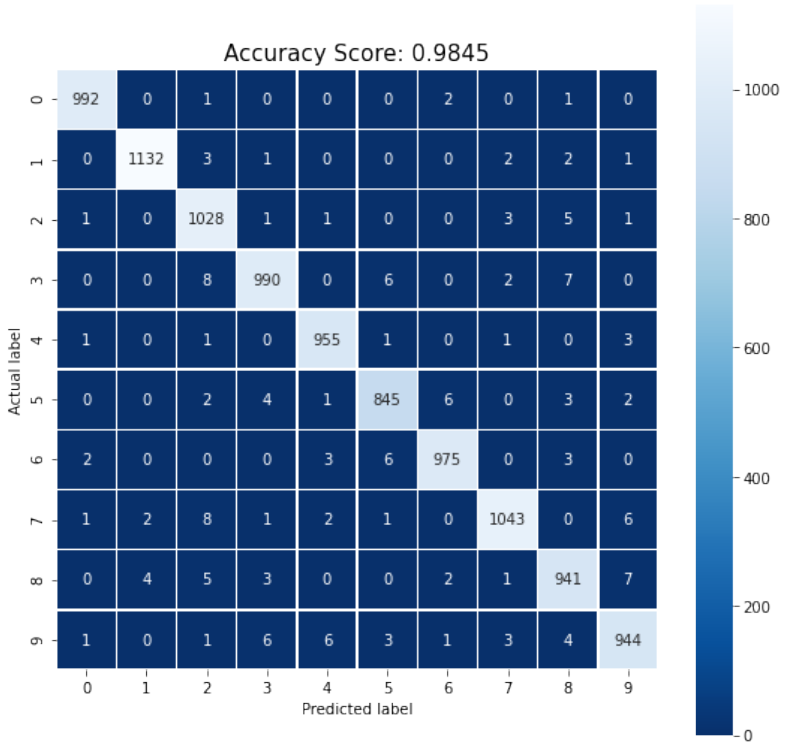
The first term is the hinge loss function and the second term is the misclassification penalty. The penalty receives increasing emphasis as λ increases. Gamma is the Kernel coefficient. We used cross-validation to find the best parameters. There is a classifier named *sklearn.svm.SVC* that implements learning based on support vectors.




```
# Learning using the best parameters and prediction_SVM
from sklearn import svm
from sklearn.model_selection import GridSearchCV
param_grid = [
#  {'C': [10, 100], 'kernel': ['linear']},
  {'C': [1, 10, 100, 1000, 10000], 'gamma': [0.1, 0.05, 0.01, 0.005, 0.001], 'kernel': ['rbf']}
  ]
t0 = time()
svm_clsf = svm.SVC()
grid_clsf = GridSearchCV(svm_clsf,param_grid,n_jobs=1, verbose=2,cv=10)
grid_clsf.fit(train_img_pca, train_lbl)
print("done in %0.3fs" % (time() - t0))
classifier = grid_clsf.best_estimator_
params = grid_clsf.best_params_
# results of cross-validation
pd.pivot_table(pd.DataFrame(grid_clsf.cv_results_),
values='mean_test_score', index='param_C', columns='param_gamma')

# heatmap to display the results of cross-validation.
import seaborn as sns
import pandas as pd
```

```
pvt = pd.pivot_table(pd.DataFrame(grid_clsf.cv_results_),
values='mean_test_score', index='param_C', columns='param_gamma')
plt.figure(figsize=(6,5))
sns.heatmap(pvt, cmap = 'OrRd')

# prediction
predictions = classifier.predict(pca.transform(test_img))
# evalution
score = classifier.score(pca.transform(test_img), test_lbl)
print(score)
```

Using the best C of **100** and gamma of **0.05** fit the model, and get the final score is **0.9845**.




```
# confusion matrix
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn import metrics
cm = metrics.confusion_matrix(test_lbl, predictions)
print(cm)

plt.figure(figsize=(9,9))
sns.heatmap(cm, annot=True, fmt="1d", linewidths=.5, square = True, cmap =
'Blues_r');
plt.ylabel('Actual label');
plt.xlabel('Predicted label');
all_sample_title = 'Accuracy Score: {0}'.format(score)
plt.title(all_sample_title, size = 15);
```

4 Discussion

The linear ridge regression classifier is time-consuming, but when the dataset is non-linear, the limitation of the linear classifier results in poor performance because it is unable to capture more complex patterns. For KNN, it has no training period which means adding new data will not impact the accuracy of the algorithm; however, KNN doesn't work well with high dimensional data because high dimensionality increases the difficulty in calculating the distance in each dimension. In this dataset, the score of performance is 0.97 when I used the first 40 components, but this score is 0.95 when I used the first 100 components. Besides, KNN is sensitive to noise in the dataset because it has no penalty. Last, SVM is more effective in high dimensional spaces due to the tricky kernel; however, it can be computationally expensive.