**Ryan Timbrook**

**NetID:** RTIMBROO

**Course:** IST 707 Data Analytics

**Term:** Summer, 2019

**Assignment:** SVMs, kNN, and Random Forest for handwritting recognition

Table of Contents

[1 Introduction 4](#_Toc17914710)

[1.1 About the Data 4](#_Toc17914711)

[1.2 Dataset Info 4](#_Toc17914712)

[1.3 Data Exploration 5](#_Toc17914713)

[1.3.1 Random Sampling of Training Dataset Images 5](#_Toc17914714)

[1.4 Data Transformation 6](#_Toc17914715)

[1.4.1 Data Normalization 6](#_Toc17914716)

[2 Model - Support Vector Machine 7](#_Toc17914717)

[2.1 Analysis and Models 7](#_Toc17914718)

[2.1.1 Data Preprocessing 7](#_Toc17914719)

[2.2 Model - Support Vector Classification 7](#_Toc17914720)

[2.2.1 Model Details 7](#_Toc17914721)

[3 Model - KNeighbors Classifier 10](#_Toc17914722)

[3.1 Analysis and Models 10](#_Toc17914723)

[3.1.1 Data Preprocessing 10](#_Toc17914724)

[3.2 Model - kNN 10](#_Toc17914725)

[3.2.1 Model Details 10](#_Toc17914726)

[4 Model - Random Forest Classification 13](#_Toc17914727)

[4.1 Model - RandomForestClassifier 13](#_Toc17914728)

[4.2 Analysis and Models 13](#_Toc17914729)

[4.2.1 Data Preprocessing 13](#_Toc17914730)

[4.3 Model - Random Forest 13](#_Toc17914731)

[4.3.1 Model Details 13](#_Toc17914732)

[5 Model Comparison 17](#_Toc17914733)

[6 Kaggle Test Result 18](#_Toc17914734)

[6.1 Submission File Format 18](#_Toc17914735)

[7 Appendix - Decision Tree (hw 6) Models 19](#_Toc17914736)

[7.1 Analysis and Models 19](#_Toc17914737)

[7.1.1 Data Preprocessing 19](#_Toc17914738)

[7.2 Model - DecisionTreeClassifier 19](#_Toc17914739)

[7.2.1 Model Details 19](#_Toc17914740)

[7.3 Model - DecisionTreeClassifier with Cross-Validation 20](#_Toc17914741)

[7.3.1 Model Details 20](#_Toc17914742)

[7.3.2 Model Parameters 20](#_Toc17914743)

[7.3.3 Model Results 21](#_Toc17914744)

[8 Appendix - Naive Bayes (hw6) Models 22](#_Toc17914745)

[8.1 Analysis and Models 22](#_Toc17914746)

[8.1.1 Data Preprocessing 22](#_Toc17914747)

[8.2 Model - Gaussian Naive Bayes Classifier 22](#_Toc17914748)

[8.2.1 Model Details 22](#_Toc17914749)

[8.2.2 Model Parameters 22](#_Toc17914750)

[8.2.3 Model Results 22](#_Toc17914751)

[8.3 Model - Complement Naive Bayes Classifier 23](#_Toc17914752)

[8.3.1 Model Details 23](#_Toc17914753)

[8.3.2 Model Parameters 23](#_Toc17914754)

[8.3.3 Model Results 23](#_Toc17914755)

[8.4 Model - Gaussian Naive Bayes Classifier with CV 24](#_Toc17914756)

[8.4.1 Model Details 24](#_Toc17914757)

[8.4.2 Model Parameters 24](#_Toc17914758)

[8.4.3 Model Results 24](#_Toc17914759)

[8.5 Model - Complement Naive Bayes Classifier with CV 25](#_Toc17914760)

[8.5.1 Model Details 25](#_Toc17914761)

[8.5.2 Model Parameters 25](#_Toc17914762)

[8.5.3 Model Results 25](#_Toc17914763)

[9 Appendix - Algorithm Performance Comparison (hw6) 26](#_Toc17914764)

[10 Appendix - Grading Rubics 27](#_Toc17914765)

# Introduction

Recognize digits 0 to 9 in handwriting images. Use the sampled data to construct prediction models using SVMs, kNN and Random Forest algorithms. Compare their performance with Naive Bayes and Decision Tree models built in week seven's homework six assignment.

The success metric is evaluated on the categorization accuracy of the predictions ( the percentage of images predicted correctly).

## About the Data

The data files train.csv and test.csv contain gray-scale images of hand-drawn digits, from zero through nine.

Each image is 28 pixels in height and 28 pixels in width, for a total of 784 pixels in total. Each pixel has a single pixel-value associated with it, indicating the lightness or darkness of that pixel, with higher numbers meaning darker. This pixel-value is an integer between 0 and 255, inclusive.

The training data set, (train.csv), has 785 columns. The first column, called "label", is the digit that was drawn by the user. The rest of the columns contain the pixel-values of the associated image.

Each pixel column in the training set has a name like pixelx, where x is an integer between 0 and 783, inclusive. To locate this pixel on the image, suppose that we have decomposed x as x = i \* 28 + j, where i and j are integers between 0 and 27, inclusive. Then pixelx is located on row i and column j of a 28 x 28 matrix, (indexing by zero).

For example, pixel31 indicates the pixel that is in the fourth column from the left, and the second row from the top, as in the ascii-diagram below.

Visually, if we omit the "pixel" prefix, the pixels make up the image like this:

000 001 002 003 ... 026 027

028 029 030 031 ... 054 055

056 057 058 059 ... 082 083

| | | | ... | |

728 729 730 731 ... 754 755

756 757 758 759 ... 782 783

The test data set, (test.csv), is the same as the training set, except that it does not contain the "label" column.

## Dataset Info

* Kaggel full training dataset has 42,000 rows and 785 columns
* Kaggel full testing dataset has 28,000 rows and 785 columns
* Kaggel subsampled training dataset has 1,400 rows and 785 columns
* Kaggel subsample testing dataset has 1,000 rows and 785 columns

For this exercise, the full training and testing datasets were used.

## Data Exploration

There were no NaN fields in the either the kaggel full training or kaggel full testing datasets.

Number of class labels: 10

Number of pixels: 784

Full training dataset info:

RangeIndex: 42000 entries, 0 to 41999

Columns: 785 entries, label to pixel783

dtypes: int64(785)

memory usage: 251.5 MB

Full testing dataset info:

RangeIndex: 28000 entries, 0 to 27999

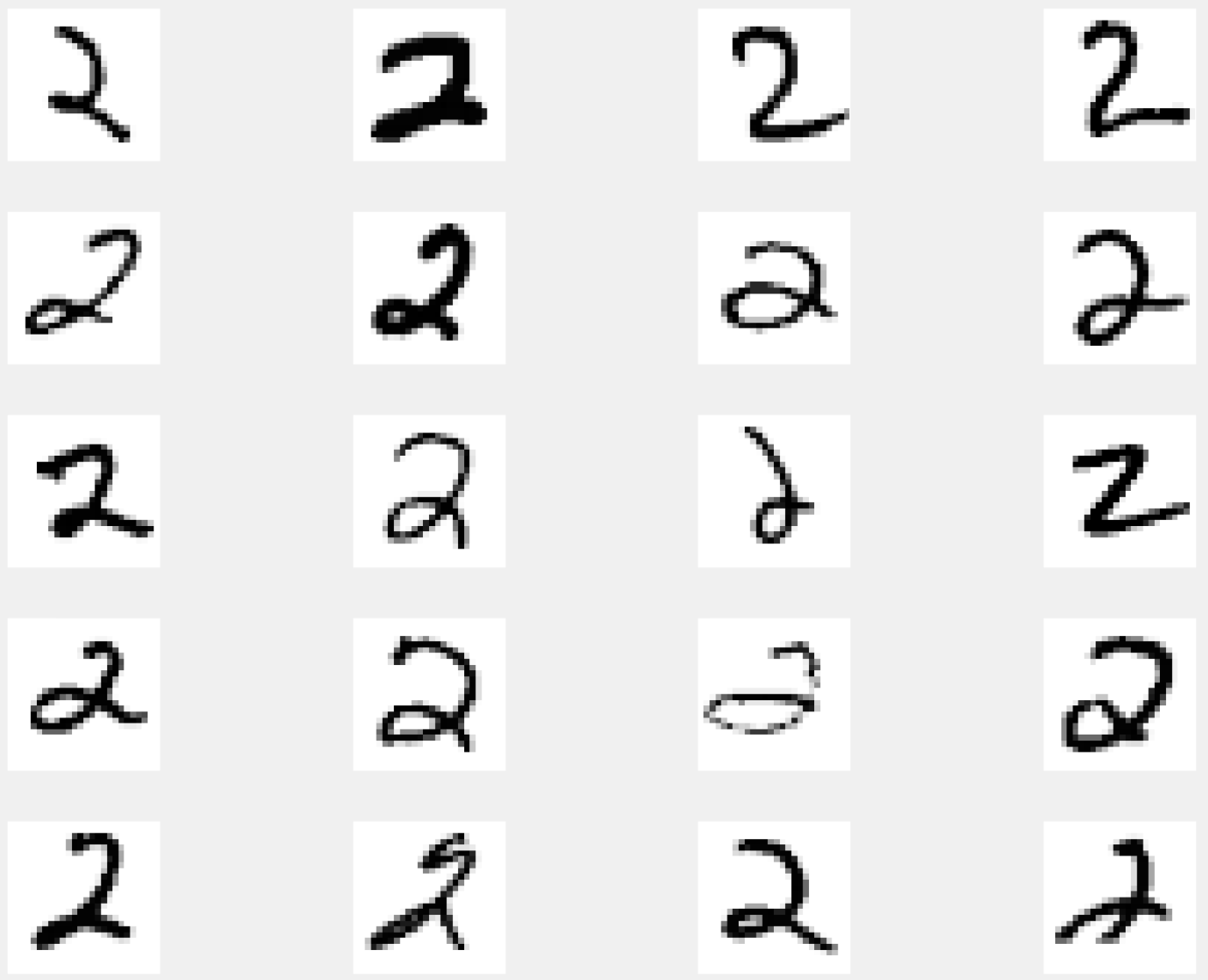
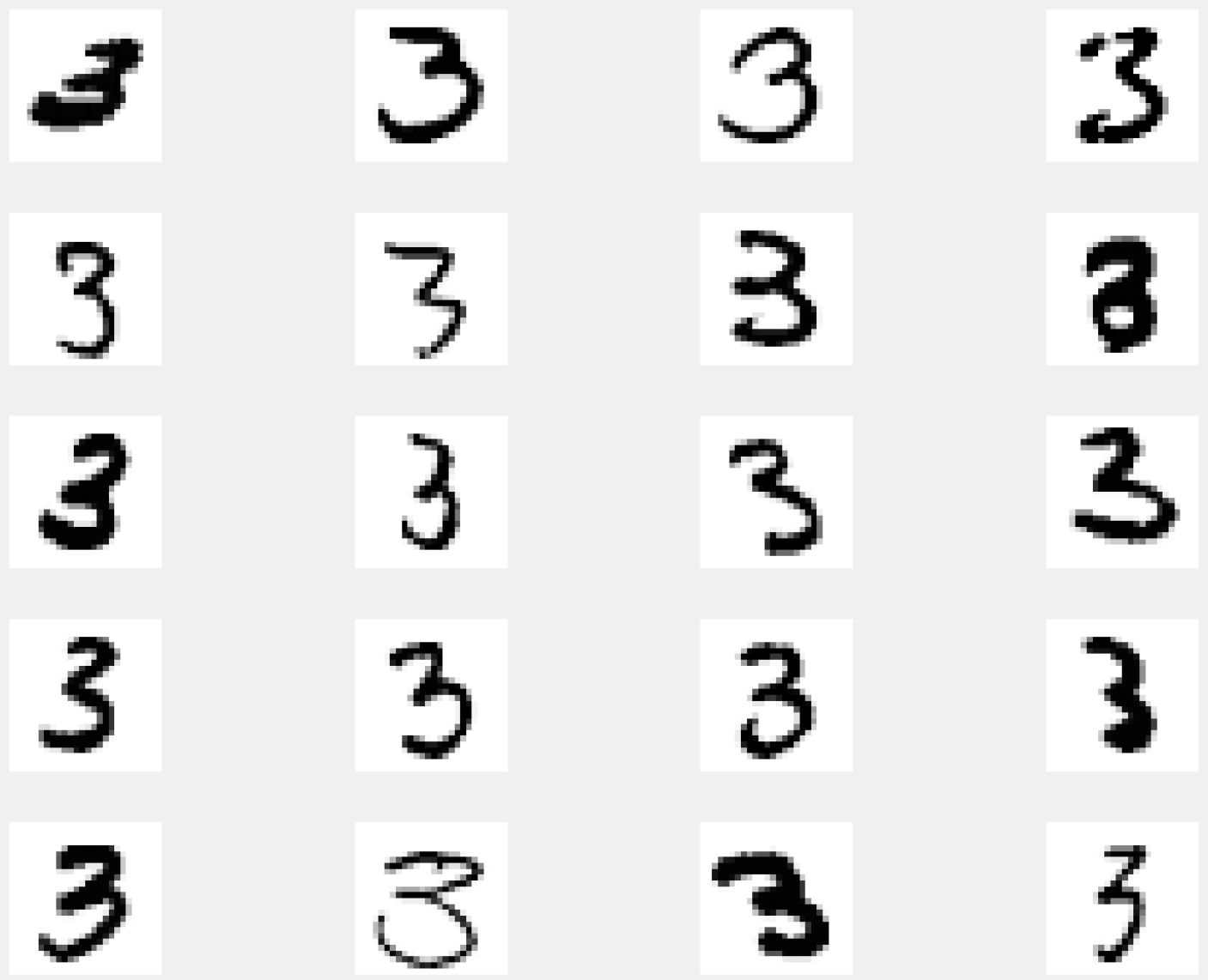
Columns: 785 entries, label to pixel783

dtypes: int64(784), object(1)

memory usage: 167.7+ MB

### Random Sampling of Training Dataset Images

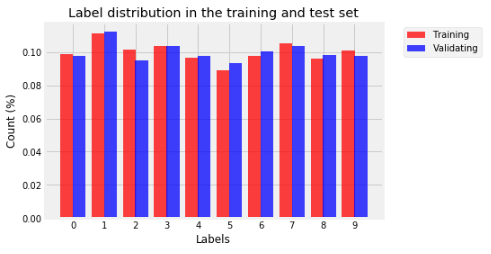
Figure: Sample Training Digit Images Figure: Sample Digit Variations



## Data Transformation

The full training dataset was split into training/validation (80/20 split) sets for model validation and prediction accuracy measurement. The test dataset provided does not have labeled classifications to be used for model validation. It is the unseen data used by the Kaggel competition for submission of predicted classification. In section five, this unseen dataset is used for the competition submission predicitons.

The training/testing datasets created from splitting the full training dataset have nearly equal number of each class label. These datasets are considered balanced.



### Data Normalization

The 'label' attribute was encoded to a catagorical, nominal variable for class classification training/testing. This was done using the sklearn.preprocessing.LabelEncoder class. Encode labels with value between 0 and n\_classes-1.

Training and test dataset's were converted to float32 objects to minimizing memory computation needs and normalized from RGB color to black and white (0-1). This was accomplished by dividing the training and test data by 255.

# Model - Support Vector Machine

Python Package: scikit-learn v0.21.3 [sklearn.svm.SVC](https://scikit-learn.org/stable/modules/generated/sklearn.svm.SVC.html#sklearn.svm.SVC)

## Analysis and Models

### Data Preprocessing

Data preprocessing for this model is described above in section 1.4.

## Model - Support Vector Classification

Python Package: scikit-learn v0.21.3 [sklearn.svm.SVC](https://scikit-learn.org/stable/modules/generated/sklearn.svm.SVC.html#sklearn.svm.SVC)

SVMs are a set of supervised learning methods used for classification, regession and outliers detection. For this implementation we are using it as a multi-class classifier.

### Model Details

C-Support Vector Classification. The implementation of this class is based on [libsvm](https://www.csie.ntu.edu.tw/~cjlin/papers/libsvm.pdf). The multiclass support is handled according to a one-vs-one scheme.

#### Model Parameters

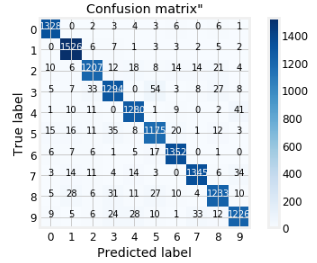
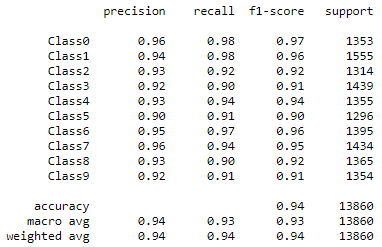
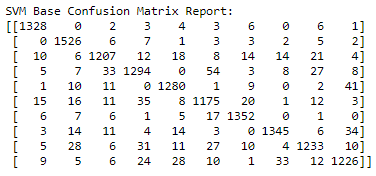
For this model, all default parameters were selected. Below is a listing of those parameters.

|  |  |
| --- | --- |
| **Parameter and Value** | **Description** |
| **C=1.0** | Penalty parameter C of the error term. |
| **cache\_size=200** | Specify the size of the kernel cache (in MB). |
| **class\_weight=None** | Set the parameter C of class i to class\_weight[i]\*C for SVC. If not given, all classes are supposed to have weight one. The “balanced” mode uses the values of y to automatically adjust weights inversely proportional to class frequencies in the input data as n\_samples / (n\_classes \* np.bincount(y)) |
| **coef0=0.0** | Independent term in kernel function. It is only significant in ‘poly’ and ‘sigmoid’. |
| **decision\_function\_shape='ovr'** | Whether to return a one-vs-rest (‘ovr’) decision function of shape (n\_samples, n\_classes) as all other classifiers, or the original one-vs-one (‘ovo’) decision function of libsvm which has shape (n\_samples, n\_classes \* (n\_classes - 1) / 2). However, one-vs-one (‘ovo’) is always used as multi-class strategy. |
| **degree=3** | Degree of the polynomial kernel function (‘poly’). Ignored by all other kernels. |
| **gamma='auto'** | Kernel coefficient for ‘rbf’, ‘poly’ and ‘sigmoid’. Current default is ‘auto’ which uses 1 / n\_features, if gamma='scale' is passed then it uses 1 / (n\_features \* X.var()) as value of gamma. |
| **kernel='rbf'** | Specifies the kernel type to be used in the algorithm. It must be one of ‘linear’, ‘poly’, ‘rbf’, ‘sigmoid’, ‘precomputed’ or a callable. |
| **max\_iter=-1** | Hard limit on iterations within solver, or -1 for no limit. |
| **probability=False** | Whether to enable probability estimates. This must be enabled prior to calling fit, and will slow down that method. |
| **random\_state=None** | The seed of the pseudo random number generator used when shuffling the data for probability estimates. If int, random\_state is the seed used by the random number generator; If RandomState instance, random\_state is the random number generator; If None, the random number generator is the RandomState instance used by np.random. |
| **shrinking=True** | Whether to use the shrinking heuristic. |
| **tol=0.001** | Tolerance for stopping criterion. |
| **verbose=True** | Enable verbose output. Note that this setting takes advantage of a per-process runtime setting in libsvm that, if enabled, may not work properly in a multithreaded context. |

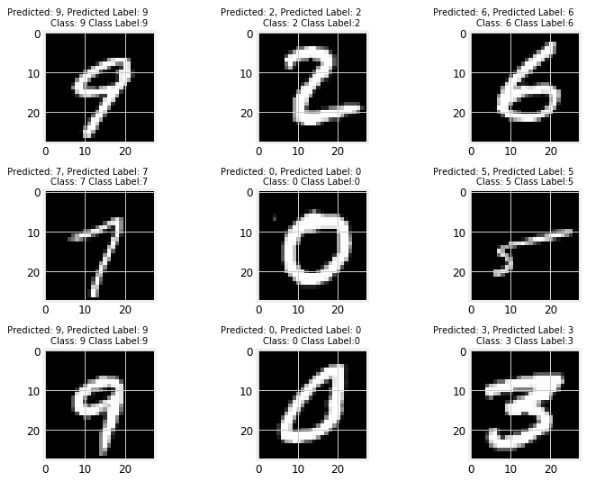
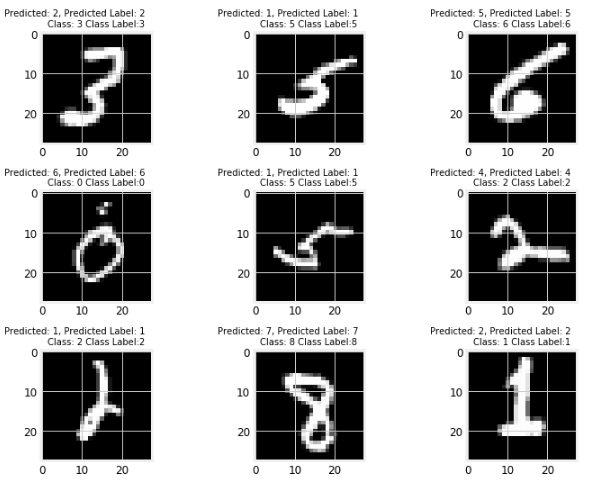
#### Model Results

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Fit Score** | **% Accuracy** | **Fit Time** | **Score Time** | **Predict Time** | **Precision** | **Recal** | **F1-Score** |
| **0.935** | 94% | 267.03898 | 189.21377 | 192.75343 | 0.94 | 0.94 | 0.94 |

**Classification Report: Confusion Matrix Report:**



Sample of Accurately Predicted Digits: Sample of Inaccurately Predicted Digits:



# Model - KNeighbors Classifier

Python Package: scikit-learn v0.21.3 [sklearn.neighbors.KNeighborsClassifier](https://scikit-learn.org/stable/modules/generated/sklearn.neighbors.KNeighborsClassifier.html#sklearn.neighbors.KNeighborsClassifier)

## Analysis and Models

### Data Preprocessing

Data preprocessing for this model is described above in section 1.4.

## Model - kNN

Classifier implementing the k-nearest neighbors vote. This classifier is well-adapted to complex, highly nonlinear datasets such as this hand written digits images dataset. The idea of K-nearest neighbors is; given a new point in the feature space, find the K closest points from the training set and assign the label of the majority of those points. In our base model, we are using the Euclidean distance metric.

An import note, is that no model is learned by a K-nearest neighbor algorithm. The classifier just stores all data points and compares any new target points with them. This is an example of instance-based learning. It is in contrast to other classifiers. This classifier is computationally intensive with a large training dataset such as the MNIST digits dataset because a large number of distances have to be computed for testing.

### Model Details

KNeighborsClassifier implements learning based on the k nearest neighbors of each query point, where k is an integer value specified at object creation as a parameter input. The optimal choice of the value k is highly data-dependent: in general a larger k suppresses the effects of noise, but makes the classification boundaries less distinct. This implementation uses basic concepts, where the nearest neighbors classification uses uniform weights. The value assigned to a query point is computed from a simple majority vote of the nearest neighbors.

kNN\_base.get\_params()

{'algorithm': 'auto',

'leaf\_size': 30,

'metric': 'minkowski',

'metric\_params': None,

'n\_jobs': None,

'n\_neighbors': 5,

'p': 2,

'weights': 'uniform'}

#### Model Parameters

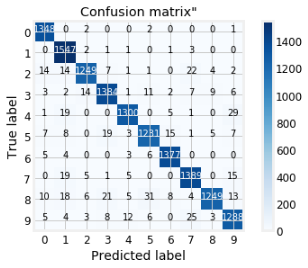
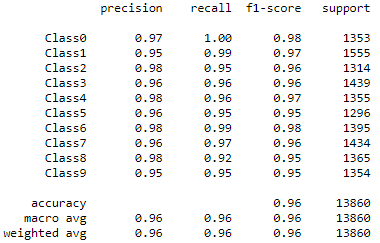
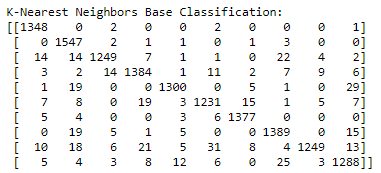
For this model, all default parameters were selected. Below is a listing of those parameters.

|  |  |
| --- | --- |
| **Parameter and Value** | **Description** |
| **n\_neighbors=5** | Number of neighbors to use by default for kneighbors queries |
| **weights="uniform"** | weight function used in prediction. ‘uniform’ : uniform weights. All points in each neighborhood are weighted equally. |
| **algorithm="auto"** | Algorithm used to compute the nearest neighbors - ‘auto’ will attempt to decide the most appropriate algorithm based on the values passed to fit method. |
| **leaf\_size=30** | Leaf size passed to BallTree or KDTree. This can affect the speed of the construction and query, as well as the memory required to store the tree. The optimal value depends on the nature of the problem. |
| **p=2** | Power parameter for the Minkowski metric. When p = 1, this is equivalent to using manhattan\_distance (l1), and euclidean\_distance (l2) for p = 2 |
| **metric="minkowski"** | The distance metric to use for the tree. The default metric is minkowski, and with p=2 is equivalent to the standard Euclidean metric. |
| **metric\_params=None** | Additional keyword arguments for the metric function. |
| **n\_jobs=None** | The number of parallel jobs to run for neighbors search. None means 1 |

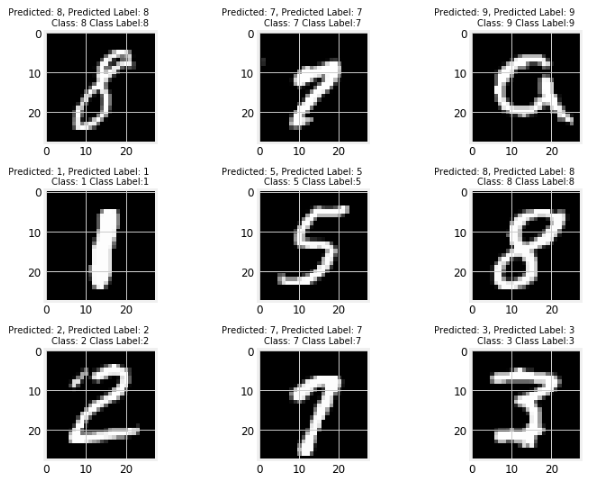
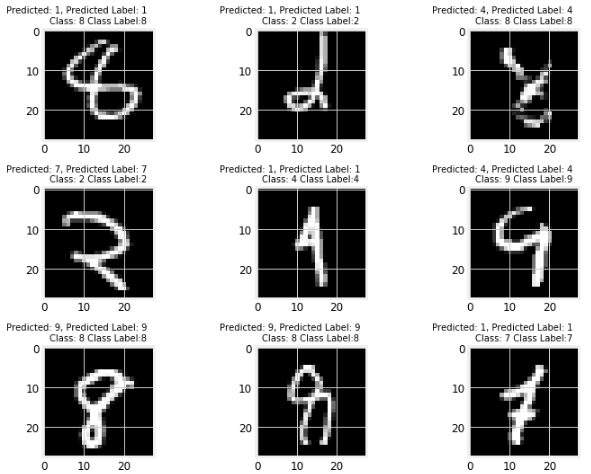
#### Model Results

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Fit Score** | **% Accuracy** | **Fit Time** | **Score Time** | **Predict Time** | **Precision** | **Recal** | **F1-Score** |
| 0.964 | 96% | 5.9340 | 609.41129 | 537.32265 | 0.96 | 0.96 | 0.96 |

**Classification Report: Confusion Matrix Report:**



Sample of Accurately Predicted Digits: Sample of Inaccurately Predicted Digits:



# Model - Random Forest Classification

## Model - RandomForestClassifier

Python Package: scikit-learn v0.21.3 [sklearn.ensemble.RandomForestClassifier](https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html)

A random forest is a meta estimator that fits a number of decision tree classifiers on various sub-samples of the dataset and uses averaging to improve the predictive accuracy and control over-fitting. The sub-sample size is always the same as the original input sample size but the samples are drawn with replacement if bootstrap=True (default).

## Analysis and Models

### Data Preprocessing

Data preprocessing for this model is described above in section 1.4.

## Model - Random Forest

This algorithm uses a perturb-and-combine techniques specifically designed for trees. A diverse set of classifieres are created by introducing randomness in the classifier construction. The prediction of the ensemble is given as the averaged prediction of the individual classifiers. Like decision trees, forests of trees also extend to multi-output problems. In random forests, each tree in the ensemble is built from a sample drawn with replacement. When splitting each node during the construction of a tree, the best split is found either from all input features or a random subset of size max\_features.

### Model Details

rf\_base.get\_params(deep=True):

{'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': 100,

'n\_jobs': None,

'oob\_score': False,

'random\_state': None,

'verbose': 2,

'warm\_start': False}

#### Model Parameters

For this model, all default parameters were selected. Below is a listing of those parameters.

For parameter tuning techniques, see [this](https://scikit-learn.org/stable/modules/ensemble.html#random-forest-parameters) for more details.

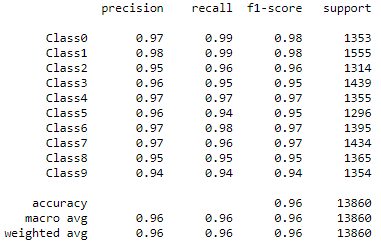
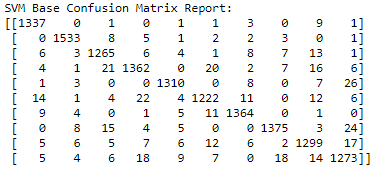
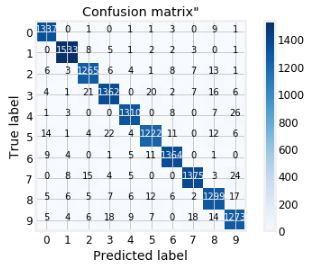
|  |  |
| --- | --- |
| **Parameter and Value** | **Description** |
| **n\_estimators=100** | The number of trees in the forest. |
| **criterion="gini"** | The function to measure the quality of a split. Supported criteria are “gini” for the Gini impurity and “entropy” for the information gain. |
| **max\_depth=None** | The maximum depth of the tree. If None, then nodes are expanded until all leaves are pure or until all leaves contain less than min\_samples\_split samples |
| **min\_samples\_split=2** | The minimum number of samples required to split an internal node |
| **min\_samples\_leaf=1** | The minimum number of samples required to be at a leaf node. A split point at any depth will only be considered if it leaves at least min\_samples\_leaf training samples in each of the left and right branches |
| **min\_weight\_fraction\_leaf=0.0** | The minimum weighted fraction of the sum total of weights (of all the input samples) required to be at a leaf node. Samples have equal weight when sample\_weight is not provided. |
| **max\_features="auto"** | The number of features to consider when looking for the best split: If “auto”, then max\_features=sqrt(n\_features). |
| **max\_leaf\_nodes=None** | Grow trees with max\_leaf\_nodes in best-first fashion. Best nodes are defined as relative reduction in impurity. If None then unlimited number of leaf nodes. |
| **min\_impurity\_decrease=0.0** | A node will be split if this split induces a decrease of the impurity greater than or equal to this value. |
| **min\_impurity\_split=None** | Threshold for early stopping in tree growth. A node will split if its impurity is above the threshold, otherwise it is a leaf. |
| **bootstrap=True** | Whether bootstrap samples are used when building trees. If False, the whole datset is used to build each tree. |
| **oob\_score=False** | Whether to use out-of-bag samples to estimate the generalization accuracy. |
| **n\_jobs=None** | The number of jobs to run in parallel for both fit and predict. None means 1 unless in a joblib.parallel\_backend context. -1 means using all processors. |
| **random\_state=None** | if int, random\_state is the seed used by the random number generator; If RandomState instance, random\_state is the random number generator; If None, the random number generator is the RandomState instance used by np.random |
| **verbose=2** | Controls the verbosity when fitting and predicting. |
| **warm\_start=False** | When set to True, reuse the solution of the previous call to fit and add more estimators to the ensemble, otherwise, just fit a whole new forest. |
| **class\_weight=None** | Weights associated with classes in the form {class\_label: weight}. If not given, all classes are supposed to have weight one. For multi-output problems, a list of dicts can be provided in the same order as the columns of y. |

The main parameters to adjust when using these methods is n\_estimators and max\_features. The former is the number of trees in the forest. The larger the better, but also the longer it will take to compute. Also, the results will stop getting significantly better beyone a critical number of trees. The later is the size of the random subsets of features to consider when splitting a node. The lower the greater the reduction of variance, but also the greater the increase in bias.

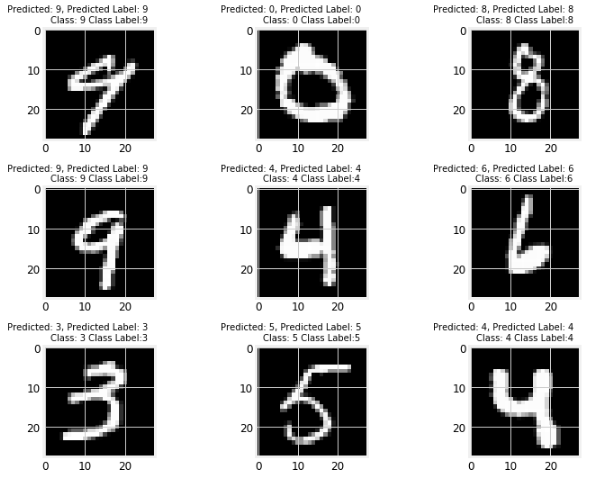
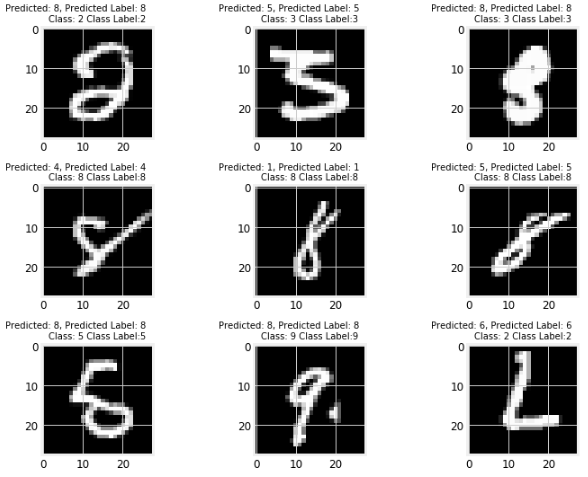
#### Model Results

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Fit Score** | **% Accuracy** | **Fit Time** | **Score Time** | **Predict Time** | **Precision** | **Recal** | **F1-Score** |
| **0.9624** | 96% | 25.2947 | 0.8023 | 0.8072 | 0.96 | 0.96 | 0.96 |

**Classification Report: Confusion Matrix Report:**



Sample of Accurately Predicted Digits: Sample of Inaccurately Predicted Digits:



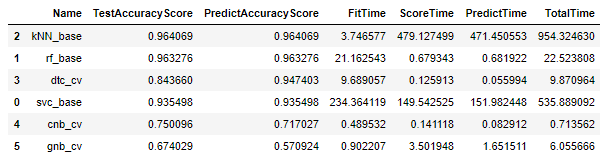
# Model Comparison

Recognize digits 0 to 9 in handwriting images. Use the sampled data to construct prediction models using SVMs, kNN and Random Forest algorithms. Compare their performance with Naive Bayes and Decision Tree (cnb\_cv, gnb\_cv, dtc\_cv) models built in week seven's homework six assignment.

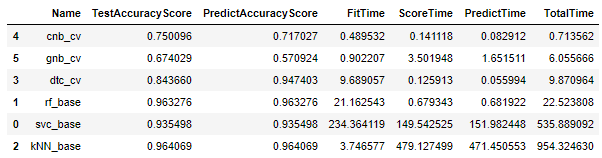
The success metric is evaluated on the categorization accuracy of the predictions ( the percentage of images predicted correctly).

Overall, the KNeighbors Classifier had a marginally higher accuracy score then the Random Forest Classifier making it the best predictor model of this dataset. That is if compute time isn't a choice factor. Due to that the kNN classifier is an instance-based learner, it takes a much longer compute time, when scoring and predicting datasets, then other classifiers. Compared with the Naive Bayes and Decision Tree models, these new models, SVMs, kNNs, and Random Forests, all out performed them significantly. Most likely this is due to the hyperparameter configurations of each model used during this round of testing. For simple baseline comparison, mostly default parameters were used. Further trials should be conducted testing various hyperparamter settings to see if any change this outcome.

*Model Performance Ranking Table, sorted by PredictAccuracyScore descending:*



*Model Performance Ranking Table, sorted by FitTime, ScoreTime, PredictTime ascending:*



# Kaggle Test Result

## Submission File Format

The submission file should be in the following format: For each of the 28000 images in the test set, output a single line containing the ImageId and the digit predicted. For example, if predict that the first image is of a 3, the second image is of a 7, and the third image is of a 8, then the submission file would look like:

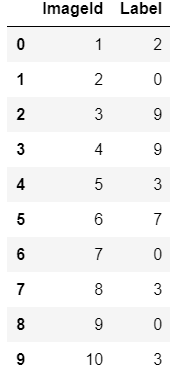
ImageId,Label  
1,3  
2,7  
3,8

(27997 more lines)

The evaluation metric for this contest is the categorization accuracy, or the proportion of test images that are correctly classified. For example, a categorization accuracy of 0.97 indicates that, all but 3% of the images have been correctly classified.

Based on the above best accuracy model, KNeighbors Classifier, the associated kNN\_submission.csv file was generated using the kNN model to predict class labels of the held out kaggel test dataset.

Sample submission head:



# Appendix - Decision Tree (hw 6) Models

## Analysis and Models

### Data Preprocessing

To minimize memory consumption, the datasets attributes were reduced to int32 objects from int64.

## Model - DecisionTreeClassifier

Python package: [scikit-learn sklearn.tree.DecisionTreeClassifier](https://scikit-learn.org/stable/modules/generated/sklearn.tree.DecisionTreeClassifier.html)

### Model Details

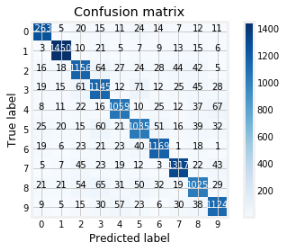
This DecisionTreeClassifier represents a baseline measurement for the Cross-Validation model detailed in seciont 2.3. The model is fit (trained on) the training dataset described in section 1.1.2 above. (i.e., train\_test\_split method)

#### Model Parameters

As a baseline model, all default parameters were used in it's build. As such, the default values for the parameters controlling the size of the trees (max\_depth, min\_samples\_leaf) lead to fully grown and unpruned trees; which on this data set lead to very large trees.

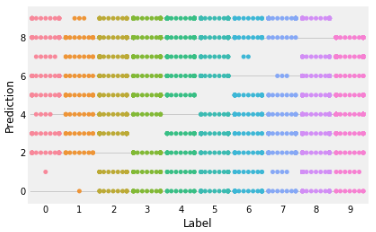
To obtain a deterministic behavior during fitting, random\_state was fixed to 0.

#### Model Results



* Fit Score: [0.84726] - 84.7%
* Fit Time: [9.80047]
* Score Time: [0.05957]
* Predict Time: [0.10187]
* Percent Accurately Labeled: [0.84726]

Misclassified Labels:



## Model - DecisionTreeClassifier with Cross-Validation

To avoid the risk of overfitting on a standard test set, cross-validation is used as a means of subdividing the training dataset to hold out a validation set. Training proceeds on the training set, after which evaluation is done on the validation set, when the trials are complete and seem successful, final evaluation is done on the held out test set. In this approach, called k-fold CV, the training set is split into k smaller sets. A model is trainined using k-1 of the folds as training data; the resulting model is validated on the remaining part of the data. (i.e., it is used as a test set to compute a performance measure such as accuracy).

### Model Details

This is a DecisionTreeClassifier with Cross-Validation. Training and validation are performed on the full training dataset (i.e., without train\_test\_split methods) using cross-validation procedures. For final evaluation, the held out test set from the train\_test\_split procedures is used for prediction accuracy measurement. The held out kaggle full test dataset is used for final prediction and competition submission.

### Model Parameters

The DecisionTreeClassifier object is instantiated with default parameters and random\_state equal to 0. Cross validation is used for training and scoring the model with cross validation parameters set to 3. This created three experiments. Overall execution time to process this cross validation procedure with three folds was 33.63 seconds.

Note that cross validation parameters, return\_train\_score and return\_estimator were set to True to report on performance metrics. These parameters increase process execution time by 10 plus seconds.

### Model Results

Three fold, Cross Validation Metrics:

Fit Time: [11.02870226, 10.70894098, 10.53186178]

Score Time: [0.11198735, 0.10233521, 0.1799221 ]

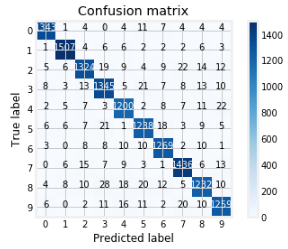
Test Recall Scores: [0.84261269, 0.84340589, 0.84338957]

Test Precision Scores: [0.84285837, 0.8436598, 0.84341631]

Train Recall Scores: [1., 1., 1.]

Train Precision Scores: [1., 1., 1.]

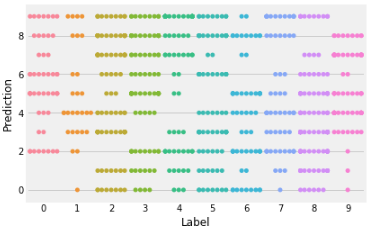
For comparision to the baseline DecisionTreeClassifier, the experment with the best test precision recall was used to predict class labels on the test dataset from the train\_test\_split procedures.



It's percent accuracy on class labeling was: [0.94898]

\*The 10% increase in accuracy is most likely due to overfitting.

Misclassified Labels:



# Appendix - Naive Bayes (hw6) Models

*T*he [**sklearn.naive\_bayes**](https://scikit-learn.org/stable/modules/classes.html#module-sklearn.naive_bayes) module implements Naive Bayes algorithms. These are supervised learning methods based on applying Bayes’ theorem with strong (naive) feature independence assumptions.

## Analysis and Models

### Data Preprocessing

No additional data preprocessing we done for the Naive Bayes Models.

## Model - Gaussian Naive Bayes Classifier

[**GaussianNB**](https://scikit-learn.org/stable/modules/generated/sklearn.naive_bayes.GaussianNB.html#sklearn.naive_bayes.GaussianNB) implements the Gaussian Naive Bayes algorithm for classification. The likelihood of the features is assumed to be Gaussian:

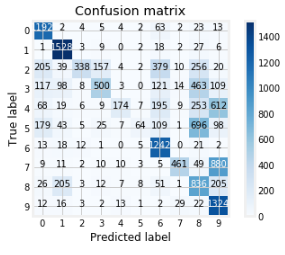
### Model Details

This Gaussian Naive Bayes Classifier represents a baseline measurement for the Cross-Validation model detailed in section 3.4. The model is fit (trained on) the training dataset described in section 1.1.2 above. (i.e., train\_test\_split method)

### Model Parameters

As a baseline model, all default parameters were used in it's build.

### Model Results



* Fit Score: [0.5526] - 55.2%
* Fit Time: [0.5442]
* Score Time: [1.8223]
* Predict Time: [1.7563]
* Percent Accurately Labeled: [0.5525] - 55.2%

## Model - Complement Naive Bayes Classifier

The Complement Naive Bayes classifier was designed to correct the “severe assumptions” made by the standard Multinomial Naive Bayes classifier. It is particularly suited for imbalanced data sets.

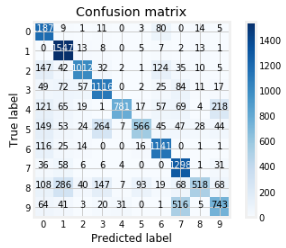
### Model Details

This Complement Naive Bayes Classifier represents a baseline measurement for the Cross-Validation model detailed in section 3.5 as well as a comparison to the Gaussian Naive Bayes Classifier. The model is fit (trained on) the training dataset described in section 1.1.2 above. (i.e., train\_test\_split method)

### Model Parameters

As a baseline model, all default parameters were used in it's build.

### Model Results



* Fit Score: [0.714935] - 71.5%
* Fit Time: [0.192643]
* Score Time: [0.079123]
* Predict Time: [0.076702]
* Percent Accurately Labeled: [0.71493]

## Model - Gaussian Naive Bayes Classifier with CV

### Model Details

This is a Gaussian Naive Bayes Classifier with Cross-Validation. Training and validation are performed on the full training dataset (i.e., without train\_test\_split methods) using cross-validation procedures. For final evaluation, the held out test set from the train\_test\_split procedures is used for prediction accuracy measurement. The held out kaggle full test dataset is used for final prediction and competition submission.

### Model Parameters

The GaussianNaiveBayes object is instantiated with default parameters. Cross validation is used for training and scoring the model with cross validation parameters set to 3. This created three experiments. Overall execution time to process this cross validation procedure with three folds was 34.4 seconds.

Note that cross validation parameters, return\_train\_score and return\_estimator were set to True to report on performance metrics. These parameters increase process execution time by 10 plus seconds.

### Model Results

Three fold, cross validation results:

Fit Time: [0.90220666, 0.80869913, 0.8218224 ]

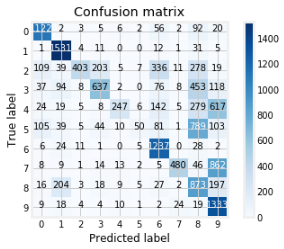
Score Time: [3.50194836, 3.44823194, 4.44742441]

Test Recall Scores: [0.55655069, 0.54957274, 0.55724842]

Test Precision Scores: [0.6740292, 0.66896641, 0.66588789]

Train Recall Scores: [0.56147661, 0.54797522, 0.56196516]

Train Precision Scores: [0.68486952, 0.67642211, 0.67696248]



Best Results:

* Fit Score: [0.67402] - 67.4%
* Fit Time: [0.90220]
* Score Time: [3.50194]
* Predict Time: [1.65151]
* Percent Accurately Labeled: [0.57092] - 57%

## Model - Complement Naive Bayes Classifier with CV

### Model Details

This is a Complement Naive Bayes Classifier with Cross-Validation. Training and validation are performed on the full training dataset (i.e., without train\_test\_split methods) using cross-validation procedures. For final evaluation, the held out test set from the train\_test\_split procedures is used for prediction accuracy measurement. The held out kaggle full test dataset is used for final prediction and competition submission.

### Model Parameters

The ComplementNaiveBayes object is instantiated with default parameters. Cross validation is used for training and scoring the model with cross validation parameters set to 3. This created three experiments. Overall execution time to process this cross validation procedure with three folds was 2.83 seconds.

Note that cross validation parameters, return\_train\_score and return\_estimator were set to True to report on performance metrics. These parameters increase process execution time by 10 plus seconds.

### Model Results

Three fold, cross validation results:

Fit Time: [0.48953247, 0.48158693, 0.45195508]

Score Time: [0.14111781, 0.14356256, 0.15300584]

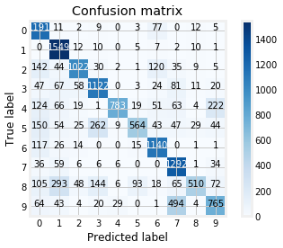
Test Recall Scores: [0.71036493, 0.70859042, 0.70432896]

Test Precision Scores: [0.75009572, 0.74860591, 0.74579701]

Train Recall Scores: [0.70791525, 0.70864266, 0.71128889]

Train Precision Scores: [0.74913705, 0.75059161, 0.75032785]

Best Results:



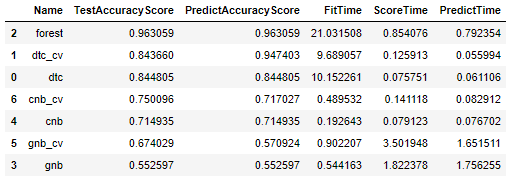
* Fit Score: [0.75009] - 75%
* Fit Time: [0.48953]
* Score Time: [0.14111]
* Predict Time: [0.08291]
* Accurately Labeled: [0.71702] - 71%

# Appendix - Algorithm Performance Comparison (hw6)

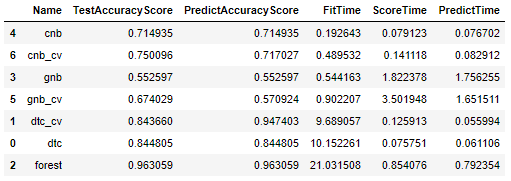
Leaving the random forest algorathm out of this comparison, overall the decission tree algorithms performed better than the naive bayes algorithms, both complement and gaussian implementations, on this data set in predicting hand written digits 0 - 9. The decision tree with three cross fold validation had a 94.7% prediction accuracy to the test dataset and an 84.3% accuracy on the cross-validation test sets. The 10% increase in prediction accuracy over test accuracy is most likely due to overfitting.

Comparing performance speeds, the naive bayes algorithms out performed the decision tree algorithms substantually. Their speeds were computed in milliseconds, while the decission tree's were in seconds, with the random forest taking the longest at 21 seconds. This would be the result of the number of expermiments produced by the algorithms and how large the tree's could get. Without tuning the tree parameters they can grow very large.

*Model Performance Ranking Table, sorted by PredictAccuracyScore descending:*



*Model Performance Ranking Table, sorted by FitTime, ScoreTime, PredictTime ascending:*



# Appendix - Grading Rubics

Grading rubrics:

1. Are the models constructed correctly?

2. Is the result analysis conclusion convincing?

3. Is sufficient details provided for others to repeat the analysis?

4. Does the analysis include irrelevant content?

5. Successful submission to Kaggle?