Comparison of different methods for multiclass classification problems

STA2101 Project

Name: Tian Han (Eric) **Guan**

Student ID: 998978058

Department: Statistics

Program: MSc

Email: tianhan.guan@mail.utoronto.ca

Abstract

This project discusses a popular topic in both statistics and machine learning of how to predict class labels in a multiclass classification problem. The project is divided into three sections. Section 1 discusses the One-vs-All and One-vs-One method which are techniques to transform a multiclass classification problem into multiple binary classification problems. Section 2 outlines the multinomial logistic regression and discusses why it is not sufficient in many cases. Section 3 applies different multiclass classification methods to the Letter Image Recognition dataset, and compares the model performance as well as provides some insights on each method. Programming in this project are done using Python's scikit-learn library.

1. Overview of multiclass classification problems

As an extension of the binary classification problem which classifies an observation into one of the two classes, multiclass classification problems deal with how to classify an observation into one of three or more classes. This project discusses two categories of multiclass classification techniques which are:

1. Transform a multiclass classification problem into multiple binary classification problems
2. Extend binary classification techniques to multiclass classification problems (also known as algorithm adaption techniques)

Now, in order to fully understand multiclass classification problems, let's take a look at some commonly used techniques from each category.

2. Transform a multiclass classification problem into multiple binary classification problems

The key idea of this category of techniques is to reduce a multiclass classification problem into *multiple* binary classification problems, and then use *binary classifiers* to solve the problem. Techniques in this category can be further categorized into either 1) **One-vs-All (OVA) method** or 2) **One-vs-One (OVO) method.**

2.1 One-vs-All method

Suppose we are given a set of training data $(x_i, y_i), i = 1, \dots, n \; and \; each \; class \; label \; y_i \in \{1, \dots, K\}$ and we have selected some type of binary classifier C. Then, the One-vs-All algorithm can be illustrated by the following pseudocode:

1. For each $k = 1, 2 \ldots K,\ create\ a\ new\ label\ z_{ki}\ where$:

$$z_{ki} = \begin{cases} 1, & y_i = k\ (y_i\ is\ in\ class\ k) \\ -1, & otherwise\ (y_i\ is\ not\ in\ class\ k) \end{cases}$$

This gives the new label vector $z_k = (z_{k1}, \ldots, z_{kn}), k = 1,2,..,K$

2. Then apply the chosen binary classifier C on $(x_i, z_{ki}), i = 1,2, \ldots, n$ to obtain a new classifier f_k and this will give us *K binary classifiers* $f_1, f_2, \ldots, f_K$.

3. For a test sample *x*, the predicted label for *y* can be obtained by:

$$\hat{y} = argmax\ f_k(x)\ over\ k = 1,2, \ldots, K, where$$

$$f_k(x)\ is\ known\ as\ the\ confidence\ score\ for\ each\ classifier$$

In other words, apply all obtained binary classifiers $f_1, f_2, \ldots, f_K$ on the new data *x* and find the classifier f_k which gives the *highest confidence score*.

Remarks on the One-vs-All method:

1. This above algorithm is called One-vs-All because we are training a single binary classifier for each class by treating training samples in that class as the *positive* samples and training samples not in that class as *negative* samples. Therefore, the One-vs-All method is also known as the One-vs-Rest method.
2. Although the One-vs-All method is very easy to implement, it usually has problems when the training data has an *unbalanced or skewed class label distribution*. This is true as the binary classifiers will take more negative labels than positive labels as the positive labels are "too rare" to be seen by the classifier. This problem can be solved by applying common

techniques of *balancing the training data* such as over-sample the minority class or under-sample the majority class.

2.2 One-vs-One method

Suppose we are given a set of training data $(x_i, y_i), i = 1, \dots, n \; and \; each \; label \; y_i \in \{1, \dots, K\}$ and we have selected some binary classifier C. Then, the One-vs-One algorithm can be illustrated by the following pseudocode:

1. Create $\frac{K(K-1)}{2}$ *sub-samples* from the training data, where each sub-sample contains a *pair of classes (i, j)* from the original training data. For example, when *K=3*, there will be 3 sub-samples to be created, containing samples from classes *(1, 2), (1, 3), and (2, 3)*. For each sub-sample containing classes *(i, j)*, treating the samples from *class i* as *positive* samples (+1 as new labels) and treating the samples from *class j* as *negative* samples (-1 as new labels).

2. Based on each sub-sample, a binary classifier is obtained using the selected binary classifier C, denote it by f_{ij}. In total, $\frac{K(K-1)}{2}$ binary classifiers $f_{ij}, i = 1, .., K, j = 1, .., K, i \neq j$ will be obtained. Also note that $f_{ij} = -f_{ji}$ by definition of f_{ij}.

3. For a test sample *x*, the predicted label for *y* can be obtained by:

$$\hat{y} = argmax \left(\sum_j f_{ij}(x)\right) \; over \; i = 1,2, \dots, K$$

In other words, *all* $\frac{K(K-1)}{2}$ binary classifiers $f_{ij}'s$ are applied to the test sample *x* and the class that receives the *highest number of votes* will be selected as the predicted label.

Remarks on the One-vs-One method:

1. The One-vs-One method is also called the All-Pairs or All-vs-All method.
2. Compared to the One-vs-All method, usually it is much less sensitive to the problem of unbalanced class distribution as each binary classifier is built only on a pair of classes.

One-vs-All vs One-vs-One method

Both methods are widely used in practice due to their simplicity, and the choice of which one to use is usually based on computational efficiency. For example, if the time to build a classifier is *superlinear or exponential* in the size of the training data (for instance, Support Vector Machine), then the One-vs-One method could be a better choice as even though it needs $O(K^2)$ classifiers compared to $O(K)$ classifiers as in One-vs-All method, *each classifier* is much smaller than the classifier in One-vs-All, and therefore makes the computation much more efficient.

3. Extend binary classification techniques to multiclass classification problems

Many of the binary classification algorithms can be extended to solve multiclass classification problems, and in general these algorithms can be categorized as either a *linear* or *non-linear* classifier. This section summarizes a classical statistical method for multiclass classification problems – the *multinomial logistic regression* from both the statistics and machine learning point of view, and discusses why it is *not sufficient* in some cases.

3.1 Multinomial logistic regression

First it should be noted that there are many different ways to define the multinomial logistic regression, the following two definitions are commonly used in most literatures, with the first

definition used more widely in statistics and the second definition used more widely in machine learning.

Definition 1

One simple way to define the model is to use one of the *K* classes as the base or reference class and set up *K-1 independent binary logistic regression models* by comparing each of the *K-1* classes against the reference class. Suppose class *K* is being chosen as the reference class, then the *K-1* independent binary logistic regression models can be constructed as follows:

$$For\ each\ k = 1, \dots, K-1, ln\left(\frac{P[Y_i=k]}{P[Y_i=K]}\right) = \beta_{0,k} + \beta_{1,k}x_{i,1} + \cdots + \beta_{p-1,k}x_{i,p-1} \stackrel{\text{def}}{=} \beta_k x_i, i = 1, \dots, n$$

Define the multinomial logistic regression this way (using a *reference class* and therefore *odds ratio*) allows *both prediction and inference* to be made easily, which is important for almost all statistical methods.

Definition 2

First define the *softmax function*:

$$softmax(k, x_1, \dots, x_K) = \frac{e^{x_k}}{\sum_{j=1}^{K} e^{x_j}}\ , k = 1, \dots, K$$

Note that by taking the exponent of each x_j, the differences between them are *amplified* and therefore the softmax function will *approach 0* when x_k is *much smaller* than the maximum of all other $x_j's$ and will *approach 1* when x_k is *much larger* than all the other $x_j's$. Therefore, the softmax function can be also interpreted as a *smoothed or continuous* version of the *max function*. In other words, it can be used to *approximate* the following max function:

$$softmax(k, x_1, \dots, x_K) \approx \begin{cases} 1, & k = argmax(x_1, \dots, x_K)\ over\ k = 1, \dots, K \\ 0, & otherwise \end{cases}$$

Then the multinomial logistic regression can be defined as follows:

$$P[Y_i = k] = softmax(k, \beta_1 x_i, \dots, , \beta_K x_i), i = 1, .., n$$

Remarks

1. This definition involves no reference class (as it is not designed for inference but prediction) and will give *K equations* rather than *K-1* equations. However, since we know that $\sum_{k=1}^{K} P[Y_i = k] = 1, i = 1, \dots, n$, this means that not all $\beta_1, \dots, \beta_K$ will be *identifiable*. Therefore, one common approach is to set one of β_k's to be some constant and this is similar to the way of fixing a reference class as in the definition 1.
2. The softmax function can also be thought as a generalization of the *sigmoid function* (i.e. when *K=2*, the softmax function becomes the sigmoid function that is used in binary logistic regression).

<u>Disadvantage of multinomial logistic regression:</u>

Similar to the case of binary logistic regression, multinomial logistic regression is also a *linear classifier* and will have a linear decision boundary for the classification, this is because the inputs into the model are assumed to be *linearly separable* and the predicted value is a *linear function* of the inputs (here, the predicted *log-odds is a linear model of the inputs*). The fact that it is a linear classifier will limit logistic regression's ability to classify large complex datasets which can involve classes that are not linearly separable.

3.2 Non-linear classifiers

In practice, it is very rare for a large complex data to be linearly separable and therefore linear classifiers such as multinomial logistic regression will not be sufficient to make accurate predictions for the class labels. As a result, more flexible and complex classifiers are developed to deal with the problem. The next section compares the multinomial logistic regression with various adapted non-linear multiclass classifiers on a real dataset and see which method performs the best for the data and also get some insights on each of the method. The other classifiers included in this project includes *k-nearest neighbors (KNN), naive bayes, decision trees (random forest),* and *neural networks (multilayer perceptron (MLP)),* these classifiers are discussed with their properties as well as their advantages and disadvantages in **Appendix 1** of the report.

4. Demonstrate various multiclass classification techniques using the Letter Image Recognition dataset

4.1 Brief description of the data

The data used in this project is the **Letter Image Recognition Data** available at the Center for Machine Learning and Intelligent Systems at the University of California, Irvine public website. The data contains **20,000** rectangular pixel images where each image (observation) is **classified** as one of the **26 capital letters** (therefore this is a **26-class classification problem**) in the English alphabet. In this data, each observation has **16 attributes or features**, where each feature is either a statistical moment or edge counts that has already been **scaled** into a range of **integer values from 0 to 15**. A detailed description of each feature can be found in **Appendix 2**. Finally, the dataset has **no missing values** for all 20,000 observations.

The goal of the following analyses is to compare different classification techniques in terms of their ***predictive powers*** on this letter recognition dataset. All the analyses are performed using modules from ***Python's scikit-learn library***.

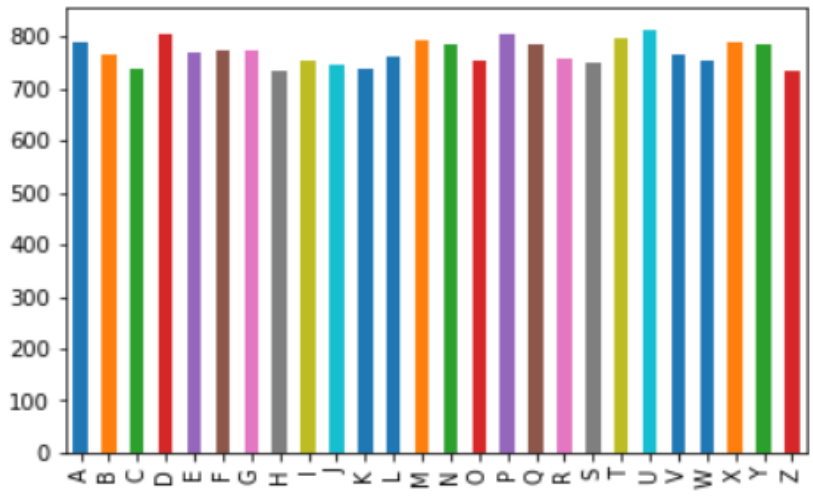
4.2 Exploratory Data Analysis (EDA)

The first step of the analysis is to perform some simple explanatory data analysis on the dataset, including the class label distribution, feature distributions, and correlation between features.

Class label distribution

From the bar plot below, it can be seen that the class labels have a roughly *uniform distribution* across all classes, as a result, there is no need to perform any *class balancing techniques* for the data. In other cases, unbalanced class labels will be a big problem for many classification techniques such as the One-vs-All method.

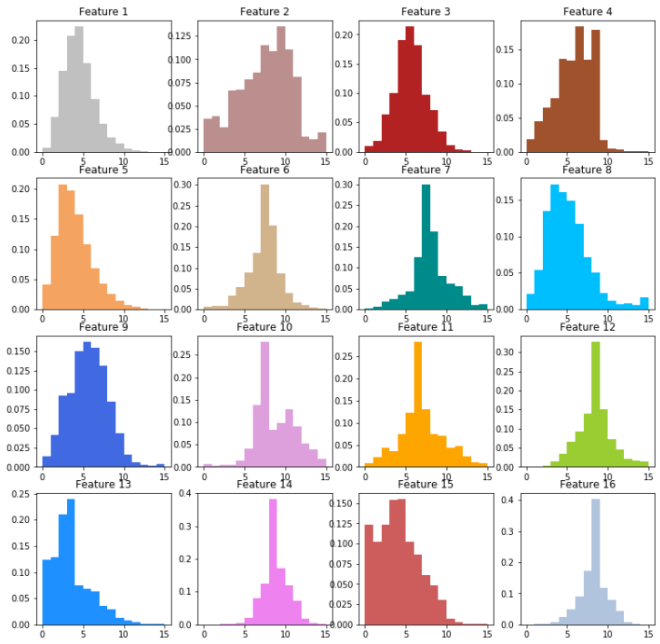
Plot 1. Class label distribution



Feature distributions

Given the fact that there are only 16 features and each feature is already been scaled, it is also interesting to see the data distribution for each feature. Plot 2 shows the data distribution of each feature using a histogram.

Plot 2. Feature distributions

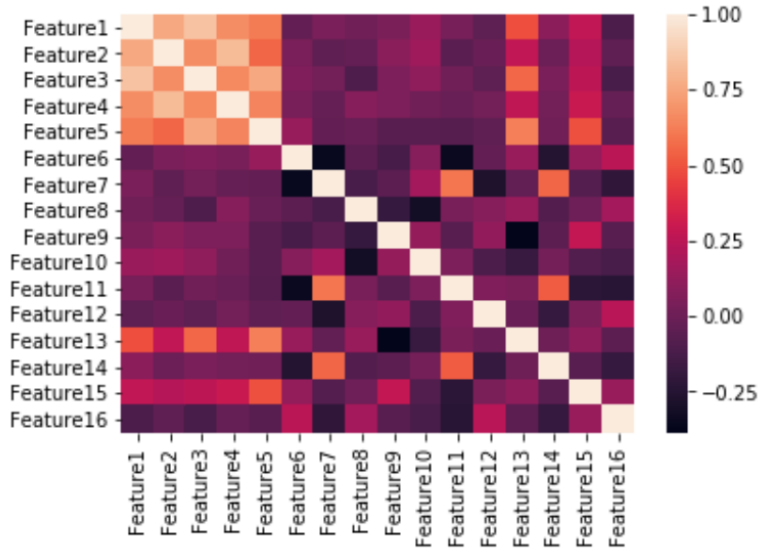


Correlation of features

Finally, it is important to check the correlation between different features. Plot 3 below shows the correlation matrix of the features using a *heat map*. It is noticed that except *feature 1-5* which have relatively strong correlations, most the features have very week correlations with each other. Given the fact that the dimension of the feature space is not very large compared to the number of observations, it is decided that all 16 features will be kept in the analysis. In general, correlated features will not improve the model performance but may raise issues related to the

curse of dimensionality such as slower computational speed, unstable results, and parameter interpretability problems.

Plot 3. Correlation matrix of features



<u>Remarks about the dataset</u>

1. The fact that this dataset is already been scaled and is free of missing values will save the analysis from the data cleaning process. Note that in other cases, it is important to scale the data first as many of the classification techniques that use *distance or similarity metrics* would require data to be *scaled* first (such as KNN and SVM).
2. It is important to recognize that the features are in *continuous or numeric values* (rather than discrete factors) from the meaning of each feature.
3. It is also noticed that the dataset is already been *shuffled randomly*. Therefore, no shuffling will be required when doing the *training-vs-testing split*. To test the model performance, the first *75%* of the dataset will be used as the *training dataset* (15,000 observations) and the remaining *25%* of the dataset will be used as the *testing dataset* (5,000 observations).

4.3 Analysis 1: Transform into multiple binary classification problems

In this section, the One-vs-All and One-vs-One method are compared in terms of their model performance. In both methods, the *non-linear SVM classifier* is used by using a *radial basis function (RBF) kernel*. **Appendix 3** briefly summarizes the basic ideas of a binary SVM. Note that although a *confusion matrix* is usually used to evaluate classifiers based on true negative and false positive rate, here, because the *number of classes is big* (26 classes), each model is simply evaluated based on its *overall accuracy rate (accuracy rate = 1- test error rate)*.

4.3.1 One-vs-All method

Results

The One-vs-All method gives an accuracy of **97.0%** on the 5,000 testing samples which seems to perform *extremely well* on this dataset.

Comments:

1. Given the fact that the data has *balanced class labels*, the One-vs-All method can be applied *directly* on the dataset. In other cases, sampling usually needs to be done before this method can be applied.
2. In general the One-vs-All method can be *slow on kernel algorithms* if the training data is huge. However, in this case, the algorithm runs very fast and has no computation issues.
3. In total, *26 non-linear SVM classifiers* have been built (*one classifier per class*).

4.3.2 One-vs-One method

Results

The One-vs-One method gives an accuracy of **97.2%** on the 5,000 testing samples which seems to perform *extremely well* on this dataset.

Comments:

1. Computational efficiency. Although in general the One-vs-One method has an advantage in speed when the classifier uses kernel algorithms (such as SVM), in this case, because the *number of classes is large*, it is *slower* than the One-vs-All method (but still runs fast for this problem).
2. In this problem, the One-vs-One method fits $\frac{26*(26-1)}{2} = 325$ non-linear SVM classifiers (*one classifier per pair of classes*).

4.4 Analysis 2: Directly using different multiclass classifiers

In this section, *5 different multiclass classification techniques* are used: *multinomial logistic regression, K-nearest neighbors, Naive Bayes, random forest, and Multilayer perceptrons*. The following section discusses the performance of each technique and provides some insights on how each technique can be tuned to provide better results.

4.4.1 Multinomial logistic regression

Compared to all the other classification algorithms, the major advantage of a multinomial logistic regression model is that it gives *meaningful interpretations of the fitted parameters* (*regression coefficients*). R's mlogit package is a powerful tool to apply multinomial logistic regression and provides many ways for inference. However, since the goal here is pure prediction, the LogisticRegression module in ***scikit-learn*** is used as it is much faster in fitting the model.

Results

The multinomial logistic regression model gives an accuracy of **75.4%** on the 5,000 testing samples which seems to *perform okay* on this dataset.

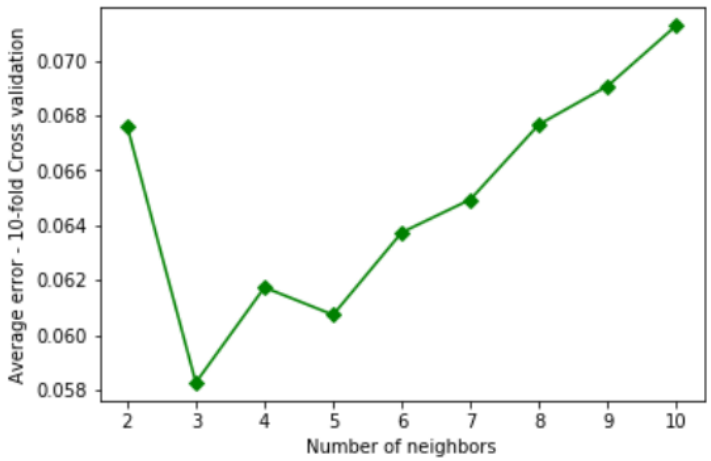
Comments:

1. As we will see later, multinomial logistic regression is not as powerful as the other classifiers and this can be due to the fact that the data here is *non-linearly separable*, and therefore using a *linear classifier* is *not sufficient* to classify the class labels.
2. Here when fitting the multinomial logistic regression model, the *multinomial cross-entropy loss function* is being used with the LBFGS solver (a type of quasi-Newton method) for the optimization. On the other hand, the training algorithm can also use the One-vs-All method where each classifier is a binary logistic regression but will follow Definition 1 of the multinomial logistic regression as mentioned before. In this case, we need 25 generalized logits to represent this 26-class problem, and in total, there will be $25 * (1 + 16) = 425$ parameters (regression coefficients).

4.4.2 K-nearest neighbors (KNN)

The first step to build a KNN classifier is to choose the *number of neighbors K* as a *hyperparameter*. In general, a *small value of K* will give a *more flexible fit* (*low bias but high variance*) and a *large value of K* will give a *less flexible fit* (*high bias but low variance*). Here, K is chosen using a *10-fold cross validation* as shown below in Plot 4. From the plot, *K = 3* is chosen as it gives the lowest error. Also note that K = 1 is excluded purposely to avoid overfitting.

Plot 4. Use k-fold cross validation to choose the number of neighbors K



Results

The KNN model gives an accuracy of **94.9%** on the 5,000 testing samples which seems to perform *very well* on this dataset.

Comments:

Although it is usually considered to be a very simple classifier, the *non-parametric nature* of a KNN model makes it a very powerful tool for this problem.

4.4.3 Gaussian Naive Bayes

A Gaussian Naive Bayes is a Naive Bayes model where the *conditional distribution* of each feature $x_i, i = 1, \dots, d,$ given the class label y follows a Gaussian distribution:

$$P(x_i|y) = \frac{1}{\sigma_y\sqrt{2\pi}} e^{-\frac{(x_i-\mu_y)^2}{2{\sigma_y}^2}}, where\ \mu_y\ and\ \sigma_y\ can\ be\ obtained\ using\ MLE$$

Remarks:

1. A Gaussian Naive Bayes seems to be a reasonable choice according to the *feature distributions* as shown in Plot 2 (as most of the features seem to be *normally distributed*).
2. The *prior distribution* $p(y)$ is automatically set to be the number of training samples in class k over the total number of training samples. Note that in this case, the prior is close to a *uniform distribution* as shown in Plot 1.
3. In total the fitted Gaussian Naive Bayes model has *26*16*2 = 832 parameters*, as the model needs a *mean and variance* parameter for *each feature per class*.

Results

The Gaussian Naive Bayes model gives an accuracy of **63.3%** on the 5,000 testing samples which *seems not to be satisfactory* on this dataset.

Comments:

As shown earlier in Plot 3, features 1-5 seem to have relatively strong correlations to each other, and this means the *naive conditional independence assumption* of the Naive Bayes model may be violated. This can partially explain the relatively unsatisfactory performance of the Gaussian Naive Bayes model here.

4.4.4. Random forest

Rather than using a single decision tree, a random forest classifier is being used in this problem. A random forest is a *meta-estimator* compared to a decision tree in the sense that it uses a large

number of decision trees to provide better predictive power by reducing the impact of over-fitting.

Remarks

1. *Number of trees* as a hyperparameter. It is commonly known that for a fixed hyperparameter configuration, increase the number of trees will not overfit the data, but may cause other hyperparameters (for example, maximum depth or number of features chosen per split) to overfit the data. In this problem, the number of trees is chosen to be *100*.
2. *Number of features to consider per split* as a hyperparameter. Given the fact some of the features are correlated, using the default of sqrt(# of features) may not give sufficient choice of features per split. As a result, the number of features to consider per split is chosen to be *8*.

Results

The random forest model gives an accuracy of **95.8%** on the 5,000 testing samples which seems to perform *very well* on this dataset.

Comments:

The fact that a random forest model has *no model assumptions* makes it a very good choice for complex multiclass classification problems such as the one here. However, it should be noted that in general a random forest is a powerful *black-box model*, that is, it is very difficult to be interpreted given its complex structure compared to a single decision tree will can usually be interpreted easily.

4.4.5. Multilayer perceptrons (MLP)

Because this is a multiclass classification problem, the *softmax activation function* is applied as the output function for the output layer, this will make sure that the output layer has *one node per class label*. Given this is a relatively large dataset, the Adam (Adaptive Moment Estimation) optimization algorithm is chosen for the MLP. The Adam algorithm is an enhanced stochastic gradient descent procedure which updates the weights iteratively based on the training data, this makes it a better choice for non-convex problems and can greatly improve the computational efficiency of the learning process.

Remarks

The only hyperparameters that are tuned in this model (for this problem) is the number of hidden layers and nodes per layer. This is usually a difficult problem as there is no standard rule of choosing them. A large number of hidden layers/nodes can create overfitting problems while a small number of hidden layers/nodes may not be sufficient to capture the patterns in the data. Here, a *single hidden layer with 100 nodes* is chosen for the MLP.

Results

The MLP model gives an accuracy of **92.3%** on the 5,000 testing samples which seems to perform *very well* on this dataset.

Comments:

Similar to the random forest classifier, a MLP model is a very good choice for complex multiclass classification problems. However, again in general neural networks are very powerful *black-box models* which makes it very difficult to draw inference from the model.

4.5 Summary of model performance for all multiclass classification techniques

The following table summarizes the predictive power of each model used in the data analysis (from most predictive to least predictive).

Method	**Prediction accuracy**
One-vs-One with SVM	**97.2%**
One-vs-All with SVM	97.0%
Random forest	95.8%
K-nearest neighbors	94.9%
Neural networks (MLP)	92.3%
Multinomial logistic regression	75.4%
Gaussian naive bayes	63.3%

In conclusion, for this Letter Image Recognition dataset, the best model (in terms of predictive power) is the One-vs-One method with SVM classifier.

Appendix 1 Summary of the non-linear multiclass classifiers used in analysis 2

1. K-nearest neighbors (KNN)

As one of the most classical non-parametric algorithms, KNN can be used directly in multiclass classification problems. As opposed to logistic regression, the major advantage of KNN is that it is a *non-linear classifier* and can therefore deal with more complex classification problems. The KNN algorithm can be summarized as follows:

Given a set of training data $(x_i, y_i), i = 1, \dots, n \; and \; each \; label \; y_i \in \{1, \dots, K\}$, suppose we want to predict the class label of a new test data *x*.

1. Select the number of neighbors to be used, denote it by *K*. Here *K* is a hyperparameter that needs to tuned.
2. Find the "closest" *K neighbors* of the new data in terms of some *distance metric D.*
3. The predicted value will be the class label that receives the *highest number of votes* by the neighbors.

Remarks on the KNN algorithm:

1. The hyperparameter *K* controls the complexity of the decision boundary and therefore choice of *K* usually faces the *bias-variance tradeoff* problem. Smaller value of *K* will result in more flexible decision boundaries (low bias & high variance), while larger value of *K* will result in less flexible decision boundaries (high bias & low variance). In practice, optimal value of *K* can be determined using error metrics such as k-fold cross validation or generalized cross validation (GCV).

2. Because a distance metric is being used to choose the neighbors, KNN is highly sensitive to the *scales* of the input data. Therefore, it is usually a good practice to *standardize* the data before using KNN.
3. It should be noted that KNN is also sensitive to *noisy samples* (such as *irrelevant and correlated* features) and the problem gets worse in high dimensional space (*curse of dimensionality*). Therefore, it is sometimes a good practice to perform dimension reduction on the data first before using KNN.

Disadvantage of KNN:

One disadvantage of KNN is its *computation inefficiency*. Suppose the features are in a d-dimensional space, then the computing time will be $O(Knd)$ which means the required number of training data increases *exponentially* with the dimension of the feature space. Although various techniques can be applied to relief this issue, KNN can be challenging in very high dimension problems.

2. Naive Bayes

Naive bayes is one of the simplest and most widely used *Bayes classifier* which can also be used directly for multiclass classification problems. A general Bayes classifier follows exactly from the Bayes' theorem:

Let $x = (x_1, .., x_d)$ *be a d dimensional vector where each* x_i *is a feature, and let* $p(C_k|x)$ represents the probability that x belongs to class k, then

$$\mathrm{p}(\mathrm{C_k}|x) = \frac{p(C_k)p(x|C_k)}{p(x)}$$

The predicted class label is obtained by taking the class label which has the *highest posterior probability*, as a result, Bayes classifier is also known as *maximum a posteriori probability (MAP) classifier*.

One problem for the Bayes classifier is that the class label likelihood function $p(x|C_k)$ can be difficult to learn from the training data. For example, one common choice is to use the multivariate Gaussian distribution, however, when the dimension of the feature space is large, the covariance matrix can contain a huge number of parameters and therefore make learning very challenging. The naive bayes classifier solves the problem by using the so-called *"naive" conditional independence assumption* which assumes that the features are conditionally independent to each other within a given class k. That is, for a given observation $x = (x_1, \dots, x_d)$, assume that $p(x_i|x_j, C_k) = p(x_i|C_k),\ \ i,j = 1, \dots d,\ \ i \neq j,\ \ k = 1, \dots, K$

Therefore, the predicted class label of *x* is:

$$\hat{y} = argmax\{p(C_k) \prod_{i=1}^{d} p(x_i|C_k)\}\ over\ k = 1,2, \dots, K$$

<u>Comparing naive bayes with logistic regression</u>

1. The key difference between a naive bayes classifier and logistic regression is that the former is a *generative model* (estimates the *joint distribution* $p(x, C)$ from the data) and the latter is a *discriminative model* (estimates $p(C|x)$ directly from the data). Because generative models estimates are based on the joint density distribution, naive bayes generally outperforms discriminative classifiers such as logistic regression when training data has a small size. When the size of the training data is large, discriminative classifiers such as logistic regression tend to perform better as they have lower asymptotic error.

2. When the class label likelihood function $p(x|C_k)$ is assumed to follow a Gaussian distribution, the naive bayes classifier is known as a Gaussian naive bayes. In this case, the naive bayes will have the same form of decision boundary (linear) as in logistic regression.

Disadvantage of the naive bayes classifier:

The major disadvantage of naive bayes occurs when *model mis-specification* occurs, that is, given a class, the features are actually correlated to each other (and this problem becomes more severe in high dimensional feature space). In this case, the prediction error can be large and discriminative models such as logistic regression might be a better choice as they are more *robust to model mis-specification issues*.

3. Random forest

Decision trees can be thought as a type of hierarchical classification algorithms that can also be used directly for multiclass classification problems. Due to their simplicity and interpretability, decision trees are very popular in classification problems. However, a single tree can be very unstable and such problem is usually solved by using a random forest model instead. Here are the major advantages of a random forest model:

- It is a type of *ensemble-learning model* in the sense that it combines multiple trees to provide better predictive performance.
- Training a random forest uses the famous technique of *bagging (bootstrap aggregating)*, which selects a sub-sample of training set with sampling with replacement to fit each tree. The random forest model applies an enhanced version of bagging, such that each split of the node only takes a *random subset of the features* (known as *feature bagging*) in order to

reduce the correlation of the trees. This greatly *reduces the variance* of the collection of trees while still enjoying the benefit of a large number of individual classifiers.

Disadvantage of random forest:

The major disadvantage of a random forest model comes in the fact that it is usually much less interpretable and therefore making it almost impossible to make inference about the problem.

4. Multilayer Perceptron (MLP)

Multiclass classification problems can be handled by a *multilayer perceptron (MLP)* – a class of *deep feedforward neural networks* which has the following properties:

- At least one hidden layer
- Fully connected – each neuron in one layer is connected with a weight w_{ij} to every node in the next layer
- Each neuron uses a *non-linear activation function* (for multiclass classification problem, often uses the *softmax activation function*), and this means that a MLP can classify data that is not linearly separable

In summary, a MLP has *K binary neurons* in the output layer so it can handle multiclass classification problem (in binary classification, it will just have one binary neuron in the output layer).

Remarks:

1. A MLP is a *non-linear discriminative classifier* that can handle very complex classification structures.

2. In summary, the value of each neuron is computed based on the *linear combination of inputs* from all neurons in the previous layer (as it is fully connected), this allows a simple MLP to represent almost all functions given appropriate parameters by using composite of simple functions to represent complex and non-linear functions. The following theorem tells this.

   Universal approximation theorem

   A feedforward neural network with a *single hidden layer and finite number of neurons* (that is, a MLP) is sufficient to approximate any continuous functions on compact subsets of the Euclidean space, under certain conditions on the activation function.

   Note: Although the theorem seems to be a strong result, it should be noted that in reality a single hidden layer MLP is usually a *high bias model*. In addition, the theorem tells nothing about the learnability of the MLP (i.e. a single hidden layer MLP may approximate such function well but may require a finite and *huge number of neurons*).
3. A MLP is a type of feedforward neural network, that is, it forms a *directed acyclic graph (DAG)* as opposed to recurrent neural networks which can have many cycles.
4. Learning of a MLP can be done by using the famous *backpropagation algorithm* which compares the output values with the targets using some error function (for example, the cross-entropy error function) and transport the error back throughout the MLP. This allows the weights to be adjusted to reduce the errors using methods such as gradient descent through many cycles until convergence.

Disadvantage of MLP:

Although a MLP can be very powerful, it is usually dangerous to be used when the size of training samples is relatively small as a complex MLP usually runs into overfitting problems. In this case, simpler methods such as KNN or logistic regression should be used.

Appendix 2 Feature information of the Letter Image Recognition dataset

	Description	**Values**
Response variable	Capital letter	A-Z (26 classes)
Feature #1	horizontal position of box	Integer 0-15
Feature #2	vertical position of box	Integer 0-15
Feature #3	width of box	Integer 0-15
Feature #4	height of box	Integer 0-15
Feature #5	total # on pixels	Integer 0-15
Feature #6	mean x of on pixels in box	Integer 0-15
Feature #7	mean y of on pixels in box	Integer 0-15
Feature #8	mean x variance	Integer 0-15
Feature #9	mean y variance	Integer 0-15
Feature #10	mean x y correlation	Integer 0-15
Feature #11	mean of x * x * y	Integer 0-15
Feature #12	mean of x * y * y	Integer 0-15
Feature #13	mean edge count left to right	Integer 0-15

Feature #14	correlation of x-ege with y	Integer 0-15
Feature #15	mean edge count bottom to top	Integer 0-15
Feature #16	correlation of y-ege with x	Integer 0-15

Appendix 3 Support Vector Machines (SVM) for binary classification

The key idea of a SVM is to separate the two classes using a (d-1)-dimensional hyperplane (if the feature space is d-dimension), where the optimal hyperplane is determined by a subset of samples (known as the support vectors) and such hyperplane maximizes the orthogonal distance from the support vectors to the hyperplane (known as maximizing the margins).

Hard-margin vs soft-margin SVM

Compared to a hard-margin SVM which always has zero training errors, a soft-margin SVM avoids overfitting by introducing a *slack variable* and includes a *penalty term C* for training sample misclassifications (note that as C approaches infinity, a soft-margin SVM will become a hard-margin SVM). Soft-margin SVMs are useful when the data is *very noisy* and therefore using a hard-margin SVM will lead to narrow margins which will overfit the training data.

Non-linear SVM classifier and the Kernel trick

If the feature space has a high dimension d, then the data is usually not linearly separable. As a result, the kernel trick is usually applied to the maximum-margin hyperplanes to obtain a non-linear SVM classifier. The idea of the kernel trick is to transform the original feature space into another high dimensional space using some non-linear kernel function such that the classifier is a

hyperplane (linear classifier) in the transformed feature space at the cost of increased testing error rate.

Appendix 4 (Python code)

1) Import data and necessary library modules

```
#Import useful modules
import os
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np
import seaborn as sns
%matplotlib inline

#Set directory and import the Letters data
os.chdir('D:\MSc2\STA2101\Project')
data = pd.read_csv('Letters.txt',header=None)
```

2) Exploratory Data Analysis

```
#Rename the columns

data.columns = ["Class"] + ["Feature" + str(num1) for num1 in range(1,17)]


#Plot the Class Label distributions and the Feature distributions
#Class label distribution
y = data.iloc[:,0]
label_dist = y.value_counts().sort_index() #create a table of counts
label_dist.plot.bar()
```

```
#Plot all 16 Feature distribution
platte = ["silver","rosybrown","firebrick","sienna","sandybrown","tan","darkcyan","deepskyblue","royal blue","plum","orange",
        "yellowgreen","dodgerblue","violet","indianred","lightsteelblue"]
f = plt.figure(figsize=(13,13))
for i in range(1,17):
        x = data.iloc[:,i]
        plt.subplot(4, 4, i)
        plt.hist(x,normed=True,bins=15,color=platte[i-1])
        plt.title('Feature ' + str(i))

#Check correlation between features using a heat map
x = data.iloc[:,1:] #The explanatory variable
sns.heatmap(x.corr())

#Now create a new column of class labels in terms of  Numbers 1-26 using the original Capital Letters A-Z
data['Class2'] = data['Class'].astype("category")
y = data['Class2'] #The response variable

#Split the dataset into training (75%) and testing (25%) dataset
train_x = x.iloc[:15000,:]
train_y = y[:15000]
test_x = x.iloc[15000:,:]
test_y = y[15000:]
```

3) Analysis 1: One-vs-All VS One-vs-One

```
#First construct a SVM classifier using the RBF kernel
from sklearn.svm import SVC
```

```
SVM = SVC(C=1,kernel='rbf',gamma='auto')
#SVM.fit(train_x,train_y)
########################
#1. One-vs-All
from sklearn.multiclass import OneVsRestClassifier
OVA = OneVsRestClassifier(SVM)
OVA.fit(train_x,train_y)
y_pred = OVA.predict(test_x)
print("The accuracy of the One-vs-All method is: ", 1-((test_y != y_pred).sum())/5000)
########################
#2. One-vs-One
from sklearn.multiclass import OneVsOneClassifier
OVO = OneVsOneClassifier(SVM)
OVO.fit(train_x,train_y)
y_pred = OVO.predict(test_x)
print("The accuracy of the One-vs-One method is: ", 1-((test_y != y_pred).sum())/5000)
```

4) Analysis 2: Multiclass classification tools

```
#Multinomial logistic regression
from sklearn.linear_model import LogisticRegression
MLR = LogisticRegression(C=1, solver='lbfgs', multi_class='multinomial')
MLR.fit(train_x,train_y)


y_pred = MLR.predict(test_x)
print("The accuracy of the multinomial logistic regression classifier is: ", 1-((test_y != y_pred).sum())/5000)
```

```
#K-nearest neighbors
from sklearn.neighbors import KNeighborsClassifier
#First choose the number of neighbors K using 5-fold cross validation
```

```
K = 5 #Number of folds
M = 10 #Maximum number of neighbors
error = []
for k in range(2,M+1):
    error2 = []
    for i in range(1,K+1):
        N = int(train_x.shape[0]/K)
        validateX = train_x.iloc[N*(i-1):N*i,:]
        validateY = train_y.iloc[N*(i-1):N*i]
        trainX = train_x.iloc[0:N*(i-1),:].append(train_x.iloc[N*i:,:])
        trainY = train_y.iloc[0:N*(i-1)].append(train_y.iloc[N*i:])
        KNN = KNeighborsClassifier(n_neighbors=k)
        KNN.fit(trainX, trainY)
        y_pred = KNN.predict(validateX)
        error2.append(((validateY != y_pred).sum())/validateY.shape[0])
    error.append(sum(error2)/len(error2))
#Plot the results
k = list(range(2, M+1, 1))
plt.plot(k,error,color='green',marker='D')
plt.xlabel('Number of neighbors')
plt.ylabel('Average error - 10-fold Cross validation')

#Then fit the KNN classifier
k = 3
KNN = KNeighborsClassifier(n_neighbors=k)
KNN.fit(train_x, train_y)
y_pred = KNN.predict(test_x)
print("The accuracy of the KNN classifier is: ", 1-((test_y != y_pred).sum())/5000)
```

```
#Gaussian Naive Bayes

from sklearn.naive_bayes import GaussianNB

gnb = GaussianNB()

gnb.fit(train_x,train_y)

y_pred = gnb.predict(test_x)

print("The accuracy of the Gaussian Naive Bayes classifier is: ", 1-((test_y != y_pred).sum())/5000)

gnb.class_prior_ #check the prior
```

```
#Random forest

from sklearn.ensemble import RandomForestClassifier

RF = RandomForestClassifier(n_estimators=10,max_features=8)

RF.fit(train_x,train_y)

y_pred = RF.predict(test_x)

print("The accuracy of the Random Forest classifier is: ", 1-((test_y != y_pred).sum())/5000)
```

```
#Multilayer perceptron

from sklearn.neural_network import MLPClassifier

MLP = MLPClassifier(solver='adam',hidden_layer_sizes=100)

MLP.fit(train_x,train_y)

y_pred = MLP.predict(test_x)

print("The accuracy of the Multilayer Perceptron classifier is: ", 1-((test_y != y_pred).sum())/5000)
```