Project 1: Prediction performance of different statistical modeling methods on analysis of the MNIST data

Rafsan Siddiqui

May 19, 2024

Abstract

In this project, I have evaluated the prediction performance of different statistical modeling methods: Multinomial Logistic Regression, Decision Trees and Random Forests, Support Vector Machine, Linear Discriminant Analysis, and Deep Neural Networks on classifying the digits of the MNIST dataset. Each model is tuned and validated to identify the most effective approach for recognizing and classifying the handwritten digits.

1. Introduction

The MNIST dataset is a benchmark in the field of statistics, machine learning and computer vision, consisting of 70,000 grayscale images of handwritten digits, each of size 28x28 pixels, with each image labeled from 0 to 9. For our exercises, we have 10,000 images' data as the training dataset, and 5,000 images' data as the testing dataset out of the original 70,000 images.

The task is to develop a model that can accurately classify the handwritten digits. The motivation behind this project is to explore and evaluate the accuracy of different statistical models in addressing the MNIST classification problem. Building on the results of a naive probabilistic model, which achieved a misclassification rate of about 22%, this project aims to push the boundaries of accuracy by employing more sophisticated models and techniques. The specific goals I've set for the project are:

1. Compare the prediction performance of various models:
   (a) Multinomial Logistic Regression (with Lasso, Ridge, and Elastic Net)
   (b) Decision Trees and Random Forests
   (c) Multi-class Support Vector Machine (SVM)
   (d) Multi-class Linear Discriminant Analysis (LDA)

(e) Deep Neural Network (DNN and CNN)

2. Describe the working procedures of the algorithms, and rationale for choosing them.
3. Optimize and tune each model to achieve the best possible classification accuracy.
4. Provide a comprehensive analysis and compare their performances.

2. Description of the Computational Algorithms

2.1. Multinomial Logistic Regression (Lasso, Ridge, and Elastic Net)

2.1.1. Overview

I have implemented Multinomial Logistic Regression with cross-validation to find the maximum classification accuracy with the best regularization parameter C.
Multinomial Logistic Regression is a suitable method for classifying the MNIST dataset for some important reasons:
1. The MNIST dataset is a multi-class classification problem. Each image is associated with one of ten possible classes (the digits 0 through 9). Multinomial Logistic Regression is designed for such multi-class classification tasks.
2. Probability Estimation: Multinomial Logistic Regression estimates the probabilities of each class for a given input, which is useful for classification tasks. For each input image, the model calculates the probability that the image belongs to each of the ten classes. The class with the highest probability is chosen as the predicted class. This probability-based approach provides a clear decision mechanism for classification.
3. Handling Multiple Classes Unlike binary logistic regression, which is limited to two classes, multinomial logistic regression can handle multiple classes directly.

For multi-class classification, the probability $P(y = k|x)$ for class k is given by the softmax function:

$$P(y = k|x) = \frac{e^{\beta_k \cdot x}}{\sum_{j=1}^{K} e^{\beta_j \cdot x}}$$

where β_k is the coefficient vector for class k, x is the feature vector, and K is the total number of classes.

We can use L1, L2 or elastic-net regularization to minimize loss functions.
For L1 regularization, the loss function is:

$$L(\beta) = -\frac{1}{N} \sum_{i=1}^{N} \log P(y_i|x_i) + \lambda \sum_{j=1}^{M} |\beta_j|$$

where N is the number of samples, λ is the regularization parameter (related to C by $\lambda = \frac{1}{C}$), and M is the number of features.

For L2 Regularization (Ridge), the objective is to minimize the following loss function:

$$L(\beta) = -\frac{1}{N}\sum_{i=1}^{N} \log P(y_i|x_i) + \lambda \sum_{j=1}^{M} \beta_j^2$$

The term $\lambda \sum_{j=1}^{M} \beta_j^2$ adds a penalty proportional to the square of the magnitude of the coefficients, which helps prevent overfitting by shrinking the coefficients.

Elastic Net regularization combines L1 and L2 penalties:

$$L(\beta) = -\frac{1}{N}\sum_{i=1}^{N} \log P(y_i|x_i) + \lambda_1 \sum_{j=1}^{M} |\beta_j| + \lambda_2 \sum_{j=1}^{M} \beta_j^2$$

2.1.2. Data Preparation and Standardization

First, the features and labels are separated from the dataset:

```
X = train_counts_df.drop(columns=['label'])
y = train_counts_df['label']
```

These would be used in all the subsequent statistical models.

The features are then standardized to have zero mean and unit variance, which is crucial for logistic regression, as it ensures that all features contribute equally to the model, preventing features with larger scales from dominating.

2.1.3. Model Training and Hyperparameter Tuning

The `LogisticRegressionCV` class is used to perform logistic regression with k-fold cross-validation to tune the regularization parameter C. The C parameter controls the strength of the regularization, with smaller values indicating stronger regularization.

The essential code snippet is:

```
model = LogisticRegressionCV(
    Cs=10,  # Number of values to try for C
    cv=5,  # Number of cross-validation folds
    penalty='l1',
    solver='saga',
    multi_class='multinomial',
    max_iter=3000,
    scoring='accuracy'
)
```

- `Cs=10`: Specifies that 10 different values of C will be tested.

- `cv=5`: Indicates that 5-fold cross-validation will be used.
- `penalty='l1'`: Uses L1 regularization (Lasso). This has been set to 'l2' and 'elasticnet' for the other regularizations.
- `solver='saga'`: An optimization algorithm suitable for large datasets and supports L1 regularization.
- `multi_class='multinomial'`: Uses the softmax function for multi-class classification.
- `max_iter=3000`: Sets the maximum number of iterations for the solver.
- `scoring='accuracy'`: The model will be evaluated based on accuracy.

The model is then trained:

```
model.fit(X_scaled, y)
```

2.1.4. Regularization Parameter Selection

During training, `LogisticRegressionCV` performs k-fold cross-validation to select the best C value. The algorithm splits the training data into k subsets, trains the model on $k-1$ subsets, and validates it on the remaining subset. This process is repeated k times, and the average performance across all folds is used to select the best C.

```
best_C = model.C_[0]
```

2.1.5. Model Evaluation

The model's cross-validation accuracy is calculated:

```
cv_score = model.score(X_scaled, y)
```

Predictions are made on the training set, and the training set accuracy is calculated:

```
y_pred = model.predict(X_scaled)
accuracy = accuracy_score(y, y_pred)
```

2.1.6. Result

Multinomial logistic regression with elastic net regularization performs the best result:

- Best C value: 166.81
- Cross-validation accuracy: 0.899
- Training set accuracy: 0.899

2.2. Decision Trees and Random Forests

2.2.1. Overview

Next, I have implemented Decision Tree and Random Forest classifiers. Both models are tuned using `GridSearchCV` for hyperparameter optimization. Here is the detailed explanation of the algorithm and how the tuning parameters are chosen.

A decision tree classifier predicts the digit by learning decision rules inferred from the features. The impurity I of a node can be measured using the Gini index or entropy:

- **Gini Impurity**:

$$I_G(t) = 1 - \sum_{i=1}^{C} p_i^2$$

- **Entropy**:

$$I_E(t) = -\sum_{i=1}^{C} p_i \log(p_i)$$

where p_i is the proportion of samples belonging to class i in the node.

A random forest is an ensemble of decision trees. Each tree is trained on a bootstrap sample of the data, and the final digit class prediction is made by taking the majority vote from all the decision trees.

2.2.2. Decision Tree Classifier

Define Hyperparameters for Tuning

```
dt_params = {
    'criterion': ['gini', 'entropy'],
    'max_depth': [None, 10, 20, 30, 40, 50],
    'min_samples_split': [2, 5, 10],
    'min_samples_leaf': [1, 2, 4]
}
```

- `criterion`: Function to measure the quality of a split. Options are "gini" for Gini impurity and "entropy" for information gain.
- `max_depth`: Maximum depth of the tree. Limits the number of levels in the tree.
- `min_samples_split`: Minimum number of samples required to split an internal node.
- `min_samples_leaf`: Minimum number of samples required to be at a leaf node.

Perform Grid Search with Cross-Validation

```
dt_grid = GridSearchCV(DecisionTreeClassifier(), dt_params, cv=5, n_jobs=-1,
        verbose=1)
dt_grid.fit(X, y)
```

- `GridSearchCV`: Performs an exhaustive search over specified parameter values using cross-validation.
- `cv=5`: Uses 5-fold cross-validation to evaluate each combination of hyperparameters.

Then I have trained the Best Model and Parameters.

Result

Fitting 5 folds for each of 108 candidates, totaling 540 fits.
Decision Tree Classifier CV Scores: [0.8135 0.807 0.7945 0.798 0.8109]
Mean CV Score: 0.8047

2.2.3. Random Forest Classifier

Define Hyperparameters for Tuning

```
rf_params = {
    'n_estimators': [50, 100, 200],
    'criterion': ['gini', 'entropy'],
    'max_depth': [None, 10, 20, 30, 40, 50],
    'min_samples_split': [2, 5, 10],
    'min_samples_leaf': [1, 2, 4]
}
```

- `n_estimators`: Number of trees in the forest.
- `criterion`: Function to measure the quality of a split (same as Decision Tree).
- `max_depth`: Maximum depth of the tree (same as Decision Tree).
- `min_samples_split`: Minimum number of samples required to split an internal node (same as Decision Tree).
- `min_samples_leaf`: Minimum number of samples required to be at a leaf node (same as Decision Tree).

Perform Grid Search with Cross-Validation

```
rf_grid = GridSearchCV(RandomForestClassifier(), rf_params, cv=5, n_jobs=-1,
        verbose=1)
rf_grid.fit(X, y)
```

Then I have trained the Best Model and Parameters.

Result

Fitting 5 folds for each of 324 candidates, totaling 1620 fits.
Random Forest Classifier CV Scores: [0.93 0.917 0.926 0.9265 0.926]
Mean CV Score: 0.925
Expectedly, Random Forest outperforms Decision Tree classifier accuracy.

2.3. Multi-class Support Vector Machine (SVM)

2.3.1. Overview

We can use Support Vector Machines (SVM) to find the hyperplane that best divides a dataset into our desired classes. For classification, SVM aims to maximize the margin between the classes, which is defined as the distance between the hyperplane and the nearest data points from either class, known as support vectors.

Mathematically, the SVM optimization problem can be formulated as:

$$\min_{\mathbf{w},b,\xi} \left(\frac{1}{2}\|\mathbf{w}\|^2 + C\sum_{i=1}^{N} \xi_i \right)$$

subject to the constraints:

$$y_i(\mathbf{w} \cdot \mathbf{x}_i + b) \geq 1 - \xi_i \quad \text{and} \quad \xi_i \geq 0 \quad \forall i$$

where:

- $\mathbf{w}$ is the weight vector.
- b is the bias term.
- ξ_i are slack variables that allow for misclassification.
- C is a regularization parameter that controls the trade-off between maximizing the margin and minimizing the classification error.

SVM can use kernel functions to transform the input space into a higher-dimensional space where a linear separator is more effective. Common kernels include linear, polynomial, and radial basis function (RBF).

I have used SVM with GridSearchCV for hyperparameter tuning to classify the MNIST dataset.

2.3.2. Define Hyperparameters for Tuning

The hyperparameters for SVM include:

- `C`: Regularization parameter values to try (0.1, 1, 10, 100).
- `gamma`: Kernel coefficient values to try (1, 0.1, 0.01, 0.001).
- `kernel`: Types of kernel to use ('linear', 'rbf', 'poly').

2.3.3. Pipeline for Scaling and SVM

A pipeline is created to standardize the features and apply the SVM:

```
pipeline = Pipeline([
    ('scaler', StandardScaler()),
    ('svc', SVC())
])
```

2.3.4. Perform Grid Search with Cross-Validation

GridSearchCV is used to perform an exhaustive search over the specified hyperparameter values using 5-fold cross-validation:

```
svm_grid = GridSearchCV(pipeline, svm_params, cv=5, n_jobs=-1, verbose=1)
svm_grid.fit(X, y)
```

Then the best hyperparameters are identified and used to train the best model.

2.3.5. Result

Fitting 5 folds for each of 48 candidates, totaling 240 fits.
SVM Classifier CV Scores: [0.9435 0.935 0.929 0.943 0.9364]
Mean CV Score: 0.937

2.4. Multi-class Linear Discriminant Analysis (LDA)

2.4.1. Overview

Next, I have used Linear Discriminant Analysis (LDA) for finding a linear combination of features that characterizes or separates the digits' classes.

LDA works by maximizing the ratio of between-class variance to the within-class variance in any particular dataset, thereby guaranteeing maximal separability. Mathematically, the objective is to project the data in a way that maximizes the distance between the means of the classes while minimizing the spread within each class.

For our multi-class classification problem, the transformation matrix W is computed by solving the following optimization problem:

$$W = \arg\max_W \frac{|W^T S_B W|}{|W^T S_W W|}$$

where:

- S_B is the between-class scatter matrix.
- S_W is the within-class scatter matrix.

LDA will classify the MNIST data by projecting the high-dimensional feature space into a lower-dimensional space while maintaining class separability.

2.4.2. Define the Pipeline

A pipeline is created to standardize the features and apply LDA:

```
pipeline = Pipeline([
    ('scaler', StandardScaler()),  # Standardize the features
    ('lda', LinearDiscriminantAnalysis())  # Apply LDA
])
```

2.4.3. Perform Cross-Validation

Cross-validation is performed to evaluate the LDA model:

```
lda_cv_scores = cross_val_score(pipeline, X, y, cv=5, n_jobs=-1)
```

- `cross_val_score`: Computes the cross-validation score for the pipeline using 5-fold cross-validation.
- `cv=5`: Uses 5-fold cross-validation.
- `n_jobs=-1`: Uses all available processors for computation.

2.4.4. Results

LDA Classifier CV Scores: [0.8495 0.8425 0.8375 0.8505 0.847]
Mean CV Score: 0.845

2.5. Deep Neural Network (DNN)

2.5.1. Overview

DNNs are capable of learning complex representations of data through multiple layers of abstraction, making them particularly effective for image classification tasks such as our MNIST dataset.

The basic building block of a neural network is the neuron, which computes a weighted sum of its inputs, applies a non-linear activation function, and passes the result to the next layer. The training process involves adjusting the weights using a technique called backpropagation, which minimizes the loss function by optimizing the weights.

Mathematically, the output of a neuron can be expressed as:

$$z = \sum_{i=1}^{n} w_i x_i + b$$

$$a = f(z)$$

where:

- w_i are the weights,
- x_i are the inputs,
- b is the bias,
- f is the activation function (e.g., ReLU, softmax),
- z is the linear combination of inputs and weights,
- a is the output of the neuron.

The loss function for a classification task with multiple classes is typically the sparse categorical cross-entropy:

$$L(y, \hat{y}) = -\sum_{i=1}^{N} y_i \log(\hat{y}_i)$$

where:

- y_i is the true label,
- $\hat{y}_i$ is the predicted probability for class i,
- N is the number of classes.

For our problem, the DNN can be used to classify the digits by learning complex patterns from the pixel values.

2.5.2. Define the Neural Network Model

I have defined a neural network model using Keras:

```
def create_model():
    model = Sequential()
    model.add(Dense(128, input_dim=X.shape[1], activation='relu'))
    model.add(Dropout(0.2))
    model.add(Dense(64, activation='relu'))
    model.add(Dropout(0.2))
    model.add(Dense(10, activation='softmax'))

    model.compile(loss='sparse_categorical_crossentropy', optimizer='adam',
    metrics=['accuracy'])

    return model
```

- The model consists of three dense layers with ReLU activation functions and dropout for regularization.
- The output layer uses the softmax activation function for multi-class classification.
- The model is compiled with the sparse categorical cross-entropy loss function and the Adam optimizer.

2.5.3. Wrap the Keras Model

The Keras model has been wrapped so it can be used with scikit-learn:

```
keras_model = KerasClassifier(build_fn=create_model, epochs=10, batch_size=32,
    verbose=0)
```

2.5.4. Standardize Features and Create Pipeline

A pipeline is created to standardize the features and apply the Keras model:

```
pipeline = Pipeline([
    ('scaler', StandardScaler()),  # Standardize the features
    ('keras', keras_model)  # Apply the Keras model
])
```

2.5.5. Cross-Validation

Cross-validation has been defined to evaluate the model:

```
kfold = KFold(n_splits=5, shuffle=True, random_state=42)
```

Then Cross-validation has been performed to evaluate the model.

2.5.6. Result

DNN Classifier CV Scores: [0.9285 0.9359 0.930 0.9304 0.9189]
Mean CV Score: 0.9287

2.6. Convolutional Neural Network

2.6.1. Overview

A Convolutional Neural Network (CNN) is a type of deep learning model specifically designed for processing structured grid data such as images. So, this would be perfect for our MNIST data classification problem.

The fundamental building blocks of a CNN include:

- **Convolutional Layers**: These layers apply convolutional filters to the input data to extract features. The operation is mathematically defined as:

$$(X * W)(i, j) = \sum_m \sum_n X(i + m, j + n)W(m, n)$$

  where X is the input image, W is the filter (or kernel), and $*$ denotes the convolution operation.

- **Activation Function**: The ReLU (Rectified Linear Unit) activation function is commonly used in CNNs:

$$\text{ReLU}(x) = \max(0, x)$$

  ReLU introduces non-linearity to the model.

- **Pooling Layers**: These layers perform down-sampling, reducing the dimensionality of the feature maps while preserving important information. Max pooling is a common pooling operation:

$$\text{MaxPool}(X) = \max(X)$$

- **Fully Connected Layers**: These layers are used at the end of the network to produce class scores based on the extracted features.

- **Dropout Layers**: Dropout is a regularization technique where a fraction of the neurons are randomly set to zero during training to prevent overfitting.

The objective of training a CNN is to minimize the loss function. For multi-class classification, the sparse categorical cross-entropy loss is used:

$$L(y, \hat{y}) = -\sum_{i=1}^{N} y_i \log(\hat{y}_i)$$

where y_i is the true label, $\hat{y}_i$ is the predicted probability for class i, and N is the number of classes.

Instead of the compressed mnist_train_counts used in the other methods, I have used CNN on the mnist_train_binary dataset, which consists of 28x28 grayscale images of digits. The spatial hierarchies and local dependencies in image data can be effectively captured by the convolutional layers.

2.6.2. Reshape the Input Data

The input data is reshaped to match the expected input shape of the CNN:

```
X_reshaped = X.values.reshape(-1, 28, 28, 1)
```

2.6.3. Define the CNN Model

Just like DNN, a convolutional neural network model has been defined using Keras:

```
def create_model():
    model = Sequential()
    model.add(Conv2D(32, kernel_size=(3, 3), activation='relu',
        input_shape=(28, 28, 1)))
    model.add(MaxPooling2D(pool_size=(2, 2)))
    model.add(Dropout(0.25))
    model.add(Conv2D(64, kernel_size=(3, 3), activation='relu'))
    model.add(MaxPooling2D(pool_size=(2, 2)))
    model.add(Dropout(0.25))
    model.add(Flatten())
    model.add(Dense(128, activation='relu'))
    model.add(Dropout(0.5))
    model.add(Dense(10, activation='softmax'))

    model.compile(loss='sparse_categorical_crossentropy', optimizer='adam',
        metrics=['accuracy'])
    return model
```

- The model consists of convolutional layers with ReLU activation functions, max pooling layers for down-sampling, dropout layers for regularization, and fully connected layers for classification.
- The output layer uses the softmax activation function for multi-class classification.
- The model is compiled with the sparse categorical cross-entropy loss function and the Adam optimizer.

The Keras model is wrapped so it can be used with scikit-learn:

```
keras_model = KerasClassifier(build_fn=create_model, epochs=10, batch_size=32,
    verbose=0)
```

2.6.4. Define and Perform Cross-Validation

Cross-validation is defined to evaluate the model:

```
kfold = KFold(n_splits=5, shuffle=True, random_state=42)
```

Then, Cross-validation has been performed to evaluate the model:

```
cv_scores = cross_val_score(keras_model, X_reshaped, y, cv=kfold, n_jobs=-1)
```

2.6.5. Results

CNN Classifier CV Scores: [0.9785 0.9804 0.9735 0.9775 0.9824]
Mean CV Score: 0.9784

CNN outperforms all the other methods and models on classifying the MNIST dataset due to their ability to capture spatial hierarchies and local dependencies in images efficiently. However, as mentioned earlier, the big mnist_train_binary dataset was used for CNN, and the computational complexity of CNN is rather higher than others.

3. Analysis and Comparison

3.1. Results

The table below summarizes the performance of various classification methods on the MNIST dataset, including their accuracy and misclassification rates.
Each of these give a better result than our previous Naive classification model (about 22%).

Method	**Accuracy**	**Misclassification Rate**
Multinomial Logistic Regression	0.899	11%
Decision Trees	0.8047	20%
Random Forests	0.925	7.5%
Support Vector Machine	0.937	6.3%
Linear Discriminant Analysis	0.845	15.5%
Deep Neural Network	0.9287	7.1%
Convolutional Neural Network	0.9784	2.2%

Table 3.1: Comparison of Classification Methods for MNIST Data

3.2. Analysis

Not surprisingly, the Convolutional Neural Network (CNN) outperforms all other models with a misclassification rate of only 2.2%. This superior performance can be attributed to CNN's ability to capture spatial hierarchies and local dependencies in the image data, making it highly effective for image classification tasks like MNIST.

The Deep Neural Network (DNN) also shows strong performance with a misclassification rate of 7.1%.

The Support Vector Machine (SVM) achieves an accuracy of 0.937 and a misclassification rate of 6.3%. SVMs are powerful classifiers that perform well on high-dimensional data, but they lack the spatial feature extraction capabilities of CNNs.

Random Forests achieve a misclassification rate of 7.5%. As an ensemble method, Random Forests combine the predictions of multiple decision trees, leading to better generalization than individual Decision Trees, which have a misclassification rate of 20%.

Multinomial Logistic Regression and Linear Discriminant Analysis (LDA) achieve misclassification rates of 11% and 15.5%, respectively. While these linear models are simpler and faster to train, they are less capable of capturing the complex patterns in image data compared to non-linear models like CNNs and SVMs.

3.3. Rationale for Model Selection

Given the results:

- **Use CNNs** for the best performance in image classification tasks like MNIST due to their ability to capture spatial hierarchies and local dependencies.
- **Consider DNNs** if computational resources are limited, as they still provide strong performance without the same level of complexity as CNNs.
- **Use SVMs and Random Forests** if a balance between accuracy and interpretability is needed, and when computational efficiency is a priority.
- **Use Decision Trees, Logistic Regression or LDA** only for simpler, less complex classification tasks or when quick, interpretable models are required.

Overall, CNNs are recommended for the highest accuracy in MNIST classification, but the choice of model should also consider the specific requirements and constraints of the task at hand.

4. Label Prediction

As seen from the results, CNN and SVM best classified the digits. I have tested CNN against the mnist_test_binary dataset and SVM against the mnist_train_counts dataset. Here are the results.

Method	**Test Dataset Used**	**Label Prediction Accuracy**	**Misclassification Rate**
CNN	mnist_test_binary	0.979	2.1%
Multiclass SVM	mnist_test_counts	0.944	5.56%

Table 4.1: Comparison of Classification Methods for MNIST Data

For predicting the labels for the test datasets, I have used the Multi-class SVM over CNN as CNN is specialized to classify such image data, is computationally complex, and required an elaborate grid structured original binary dataset for training and testing.

5. Conclusion

Overall, CNNs can be used for the highest accuracy in image classification projects like this one, but the choice of model should consider specific requirements such as computational resources and the need for model interpretability.