DOCUMENTATION

Data processing

```
# Reshape data and place into rows. Flatten the training and test data so each row
# consists of all pixels of an example
X_train = np.reshape(X_train, (X_train.shape[0], -1))
X_test = np.reshape(X_test, (X_test.shape[0], -1))
print(X_train.shape, X_test.shape) # X_train should be (10000, 3072) and X_test
should be (1000, 3072)
```

Before applying the KNN algorithm, the data needs to be preprocessed. In this step:

The image data for both training and test sets is reshaped into a format where each row represents a single example. This flattening process transforms the multi-dimensional image data into a one-dimensional array, making it easier for the algorithm to process.

KNN implementation & evaluation

```
# Performing KNN
classifier = KNearestNeighbor()

# Use the KNearestNeighbour classifier to do as follows:
# 1) Initialize classifier with training data
# 2) Use classifier to compute distances from each test example in X_test to every
training example
# 3) Use classifier to predict labels of each test example in X_test using k=5
y_test_pred = classifier.predict(X_test, k=5)

# Compute accuracy for k=5
num_correct = np.sum(y_test_pred == y_test)
accuracy = float(num_correct) / num_test
print('Got %d / %d correct with k=5 => accuracy: %f' % (num_correct, num_test,
accuracy))
```

This section focuses on implementing the KNN algorithm and evaluating its performance on the test data.

The KNearestNeighbor class is instantiated to perform the KNN classification.
The classifier is trained using the training data.
Predictions are made on the test data, with k=5 (number of nearest neighbors to consider).
Accuracy is calculated by comparing the predicted labels to the actual labels of the test data.

Cross-validation for Parameter Tuning:

```
# Perform 5-fold cross validation to find optimal k from choices below
num_folds = 5
k_choices = [1, 3, 5, 8, 10, 12, 15, 20, 50, 100]
X_train_folds = np.array_split(X_train, num_folds)
y_train_folds = np.array_split(y_train, num_folds)
k_to_accuracies = {}
# Loop over each value of k
for k in k_choices:
    k_to_accuracies[k] = []
    # For each fold of cross validation
    for num_knn in range(0, num_folds):
        # Split training data into validation fold and training folds
        y_train_cv = np.concatenate([y_train_folds[j] for j in range(num_folds) if 
j != num_knn])
        X_train_cv = np.concatenate([X_train_folds[j] for j in range(num_folds) if 
j != num_knn])
        X_val_cv = X_train_folds[num_knn]
        y_val_cv = y_train_folds[num_knn]
        classifier.train(X_train_cv, y_train_cv)
        # Predict labels of validation fold for given k value
        y_test_pred = classifier.predict(X_val_cv, k=k)
        num_correct = np.sum(y_test_pred == y_val_cv)
        accuracy = float(num_correct) / num_test
        k_to_accuracies[k].append(accuracy)
# Print and visualize cross-validation results
print("Printing our 5-fold accuracies for varying values of k:")
print()
for k in sorted(k_to_accuracies):
    for accuracy in k_to_accuracies[k]:
        print('k = %d, accuracy = %f' % (k, accuracy))
# Plot cross-validation accuracies
for k in sorted(k_to_accuracies):
    print('k = %d, avg. accuracy = %f' % (k, sum(k_to_accuracies[k])/5))
for k in k_choices:
    accuracies = k_to_accuracies[k]
    plt.scatter([k] * len(accuracies), accuracies)
# Plot trend line with error bars
accuracies_mean = np.array([np.mean(v) for k, v in 
sorted(k_to_accuracies.items())])
accuracies_std = np.array([np.std(v) for k, v in sorted(k_to_accuracies.items())])
plt.errorbar(k_choices, accuracies_mean, yerr=accuracies_std)
plt.title('Cross-validation on k')
plt.xlabel('k')
plt.ylabel('Cross-validation accuracy')
plt.show()
```

Parameter tuning is crucial for optimizing the performance of the KNN algorithm. Cross-validation is used here to determine the best value of the hyperparameter k.

The dataset is divided into k folds for cross-validation.
For each value of k in a predefined set, the algorithm is trained and evaluated on different folds.
The average accuracy across all folds is calculated for each value of k.
The results are printed and visualized to identify the optimal value of k.

Best Model Selection and Evaluation:

```
# Choose best value of k based on cross-validation results
best_k = 10

# Train final model with best_k and evaluate on test data
classifier = KNearestNeighbor()
classifier.train(X_train, y_train)
y_test_pred = classifier.predict(X_test, k=best_k)
num_correct = np.sum(y_test_pred == y_test)
accuracy = float(num_correct) / num_test
print('Got %d / %d correct on test data => accuracy: %f' % (num_correct, num_test,
accuracy))
```

Once the optimal value of k is determined through cross-validation, the final model is trained and evaluated on the test data.

The best value of k obtained from cross-validation (in this case, k=10) is selected.
A new KNearestNeighbor classifier is instantiated and trained with the entire training dataset.
Predictions are made on the test data using the best value of k.
Finally, the accuracy of the model on the test data is computed and printed.

Conclusion:

In summary, the KNN algorithm is implemented, evaluated, and tuned using cross-validation to find the optimal value of the hyperparameter k. The best model is selected based on cross-validation results and evaluated on unseen test data to measure its performance.