Machine Learning Coursework Report

ssysj1 20320552

December 2023

1 Introduction

This coursework centers around the critical task of breast cancer diagnosis using machine learning techniques. The primary objectives are outlined across several tasks, each contributing to the overarching goal. In Task 1, we initiate the process by loading and preprocessing the breast cancer dataset. Task 2 delves into the implementation and validation of a K-nearest neighbor (KNN) algorithm, exploring different input feature vectors and K values. Moving forward, Task 3 focuses on the design and optimization of a multi-layer perceptron (MLP) model, leveraging a comprehensive hyperparameter search. The ensuing Task 4 undertakes a comparative analysis, pitting the performances of the KNN and MLP models against each other, considering aspects such as accuracy, computational complexity, and overfitting. At the end of the report, there is a discussion section to clarify the comparative analysis and help make an informed choice between KNN and MLP models.

2 Task 1

2.1 Load Data

In task 1, I first load the data from the wdbc dataset file (WDBc.data). Each row represents an instance of the data, where the property values are separated by commas. There are 32 attributes in total, where the first attribute is the patient's ID, the second attribute is the category label (M for malignant, B for benign), and the remaining attributes are input characteristics.

To accomplish this step, I use the genfromtxt function from the NumPy library, specifying a comma as the separator, to load the data into a NumPy array. I extract the category labels and input features, where the labels are converted to binary form (M: 1, B: 0) for subsequent binary classification tasks.

```python
# Load data
import numpy as np

data_file = './wdbc.data'

data = np.genfromtxt(data_file, delimiter=',', dtype=str)
labels = data[:, 1]

# Replace 'M' with 1, 'B' with 0
labels = np.where(labels == 'M', 1, 0)

data = np.genfromtxt(data_file, delimiter=',')
features = data[:, 2:]
```

Figure 1: Load Data

2.2 Data Split

In this step, I divide the data into a training set and a test set. I used the train_test_split function from the Scikit-learn library to select 169 samples from the population as test data and the remaining 400 samples for training.

```python
# Split data
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(features, labels, test_size=169, random_state=42)


print("Training Data Shape:", X_train.shape)
print("Testing Data Shape:", X_test.shape)
```

Figure 2: Data Split

2.3 Principal Component Analysis (PCA)

Next, I apply principal component analysis (PCA) to reduce the dimensions of the original input features. I chose the reduced dimensions of 5, 10, 15 and 20. The goal of PCA is to map the original feature to the new feature space through linear transformations to preserve the main variance of the data. I use the PCA class in the Scikit-learn library for dimensionality reduction.

```python
# Dimensionality reduction by PCA
from sklearn.decomposition import PCA

dimensions = [5, 10, 15, 20]

all_X_train_pca = []
all_X_test_pca = []

for dim in dimensions:
    pca = PCA(n_components=dim)
    X_train_pca = pca.fit_transform(X_train)
    X_test_pca = pca.transform(X_test)

    all_X_train_pca.append(X_train_pca)
    all_X_test_pca.append(X_test_pca)
```

Figure 3: PCA

At the same time as I perform the dimensionality reduction operation, relevant information is printed, including the variance ratio of the interpretation and the variance ratio of the cumulative interpretation.

```
Dimension: 5
Explained Variance Ratio (first 3 components): [0.98135836 0.01651178 0.00191012]
Cumulative Explained Variance Ratio: [0.98135836 0.99787014 0.99978025 0.99990627 0.99998835]
Reduced Training Data Shape: (400, 5)
Reduced Testing Data Shape: (169, 5)

Dimension: 10
Explained Variance Ratio (first 3 components): [0.98135836 0.01651178 0.00191012]
Cumulative Explained Variance Ratio: [0.98135836 0.99787014 0.99978025 0.99990627 0.99998835 0.99999451
 0.99999855 0.99999937 0.99999975 0.9999999 ]
Reduced Training Data Shape: (400, 10)
Reduced Testing Data Shape: (169, 10)

Dimension: 15
Explained Variance Ratio (first 3 components): [0.98135836 0.01651178 0.00191012]
Cumulative Explained Variance Ratio: [0.98135836 0.99787014 0.99978025 0.99990627 0.99998835 0.99999451
 0.99999855 0.99999937 0.99999975 0.9999999  0.99999997 0.99999998
 0.99999999 0.99999999 1.         ]
Reduced Training Data Shape: (400, 15)
Reduced Testing Data Shape: (169, 15)

Dimension: 20
Explained Variance Ratio (first 3 components): [0.98135836 0.01651178 0.00191012]
Cumulative Explained Variance Ratio: [0.98135836 0.99787014 0.99978025 0.99990627 0.99998835 0.99999451
 0.99999855 0.99999937 0.99999975 0.9999999  0.99999997 0.99999998
 0.99999999 0.99999999 1.         1.         1.         1.
 1.         1.         ]
Reduced Training Data Shape: (400, 20)
Reduced Testing Data Shape: (169, 20)
```

Figure 4: PCA Dimension Reduction Results

3 Task 2

In Task 2, I design and implement a breast cancer diagnosis system using K-nearest neighbor (KNN) algorithm for classification, combined with feature vectors after dimensionality reduction in Task 1. Here are the main steps I took to accomplish our mission:

3.1 KNN Model Verification

First, I validate the KNN model using training data, taking into account different input feature vectors (original feature and feature after dimensionality reduction in task 1) and different K values (K=1, 3, 5, 7, 9). I write an evaluate_knn function to calculate the model's accuracy, precision, recall, and F1 values. For validation, I use the KNeighborsClassifier class in the Scikit-learn library.

```python
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score
import matplotlib.pyplot as plt

# Function to evaluate KNN models and return evaluation metrics
def evaluate_knn(X_train, X_test, y_train, y_test, k, dimension):
    knn = KNeighborsClassifier(n_neighbors=k)
    knn.fit(X_train[:, :dimension], y_train)

    y_pred = knn.predict(X_test[:, :dimension])

    accuracy = accuracy_score(y_test, y_pred)
    precision = precision_score(y_test, y_pred, pos_label=1)
    recall = recall_score(y_test, y_pred, pos_label=1)
    f1 = f1_score(y_test, y_pred, pos_label=1)

    return accuracy, precision, recall, f1
```

Figure 5: KNN Evaluate Function

3.2 Model Verification Result

I iterated over the different feature dimensions and K values, saved the validation results (accuracy, precision, recall, and F1 values), and stored them in a data structure.

```python
# Perform KNN with different feature dimensions and K values
results = {'dimension': [], 'k': [], 'accuracy': [], 'precision': [], 'recall': [], 'f1': []}

for i, dim in enumerate(dimensions):
    X_train_pca = all_X_train_pca[i]
    X_test_pca = all_X_test_pca[i]

    for k_value in [1, 3, 5, 7, 9]:
        accuracy, precision, recall, f1 = evaluate_knn(X_train_pca, X_test_pca, y_train, y_test, k_value, dim)

        # Save results
        results['dimension'].append(dim)
        results['k'].append(k_value)
        results['accuracy'].append(accuracy)
        results['precision'].append(precision)
        results['recall'].append(recall)
        results['f1'].append(f1)
```

Figure 6: KNN Model Validation

The following diagram shows validation results for different dimensions and K values.

	dimension	k	accuracy	precision	recall	f1
0	5	1	0.934911	0.932203	0.887097	0.909091
1	5	3	0.946746	0.934426	0.919355	0.926829
2	5	5	0.970414	0.983051	0.935484	0.958678
3	5	7	0.964497	0.966667	0.935484	0.950820
4	5	9	0.964497	0.966667	0.935484	0.950820
5	10	1	0.934911	0.932203	0.887097	0.909091
6	10	3	0.946746	0.934426	0.919355	0.926829
7	10	5	0.970414	0.983051	0.935484	0.958678
8	10	7	0.964497	0.966667	0.935484	0.950820
9	10	9	0.964497	0.966667	0.935484	0.950820
10	15	1	0.934911	0.932203	0.887097	0.909091
11	15	3	0.946746	0.934426	0.919355	0.926829
12	15	5	0.970414	0.983051	0.935484	0.958678
13	15	7	0.964497	0.966667	0.935484	0.950820
14	15	9	0.964497	0.966667	0.935484	0.950820
15	20	1	0.934911	0.932203	0.887097	0.909091
16	20	3	0.946746	0.934426	0.919355	0.926829
17	20	5	0.970414	0.983051	0.935484	0.958678
18	20	7	0.964497	0.966667	0.935484	0.950820
19	20	9	0.964497	0.966667	0.935484	0.950820
20	30	1	0.934911	0.932203	0.887097	0.909091
21	30	3	0.946746	0.934426	0.919355	0.926829
22	30	5	0.970414	0.983051	0.935484	0.958678
23	30	7	0.964497	0.966667	0.935484	0.950820
24	30	9	0.964497	0.966667	0.935484	0.950820

Figure 7: Validation Results

3.3 Visualization of Model Validation Results

3.3.1 Performance in different Dimensions

First, I plotted the variation trend of model accuracy, precision, recall rate and F1 value under different dimensions and K values. This helps to understand the impact of dimensions on model performance.

```
import matplotlib.pyplot as plt

# Plotting with matplotlib
fig, axes = plt.subplots(2, 2, figsize=(12, 10))
metrics = ['accuracy', 'precision', 'recall', 'f1']

for i, metric in enumerate(metrics):
    ax = axes[i//2, i%2]

    # Plot for each K value
    for k_value in [1, 3, 5, 7, 9]:
        k_results = results_df[results_df['k'] == k_value]
        ax.plot(k_results['dimension'], k_results[metric], label=f'K={k_value}')

    ax.set_title(metric.capitalize())
    ax.set_xlabel('Dimension')
    ax.set_ylabel(metric.capitalize())
    ax.legend()

plt.tight_layout()
plt.show()
```

Figure 8: Plot Performance Variation with Different K Values

From the image below, you can see that for each K value, whether the feature dimension is 5, 10, 15, 20, or 30, the performance metric remains almost constant, forming a horizontal line.

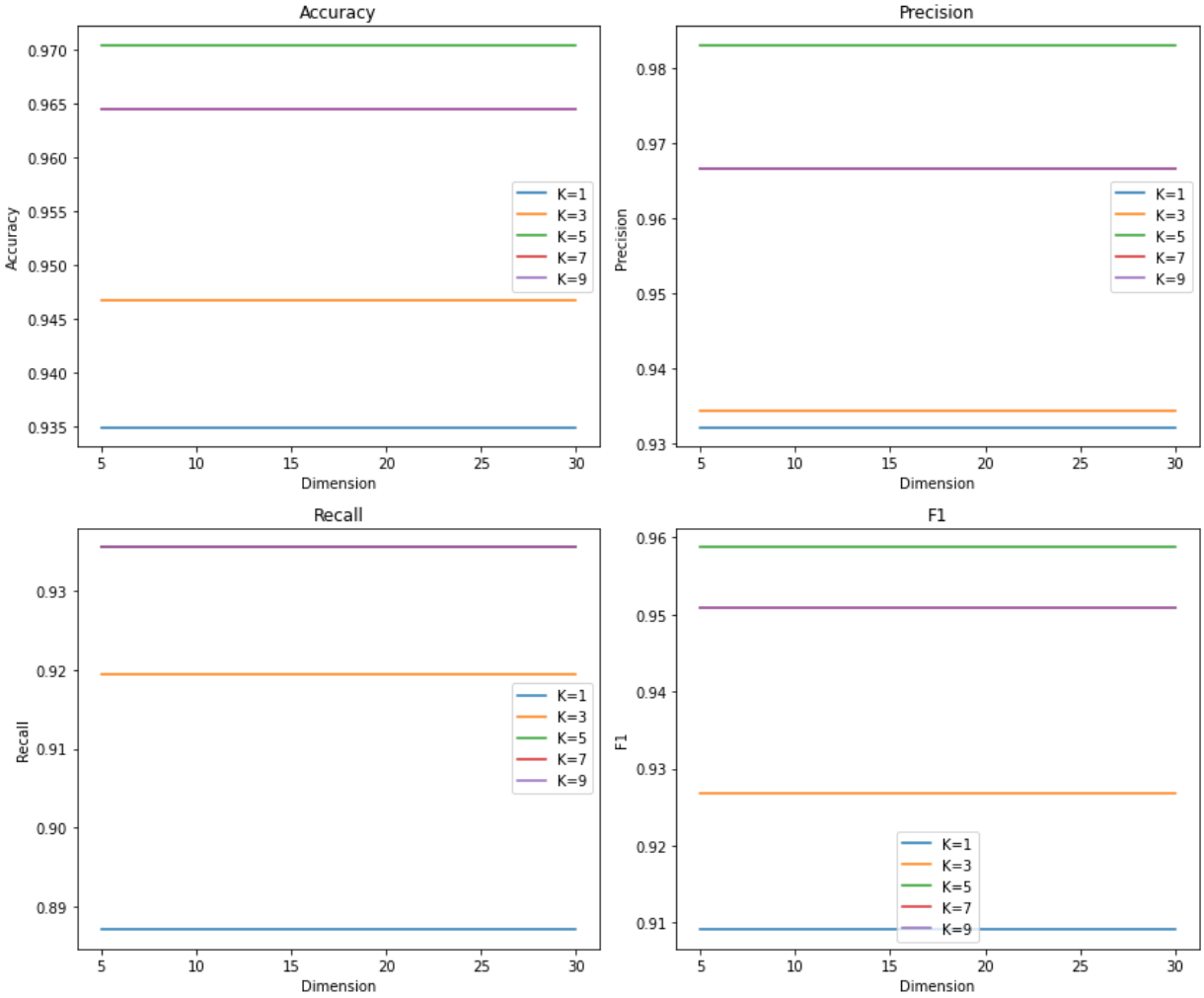

Figure 9: Performance Variation with Different K Values

The observed phenomenon can be explained in several ways:

- **Insufficient Dimensional Selection:** The reduced feature dimensions may not have captured crucial information from the data. Therefore, increasing or decreasing dimensions has a limited impact on model performance.
- **Simplicity in Data Distribution:** The data, after dimensionality reduction, might exhibit a relatively straightforward structure, resulting in similar performance of KNN in both low and high dimensions.
- **Influence of K Values:** In certain scenarios, the choice of K values may have a more significant impact on model performance, while changes in dimensions do not lead to noticeable alterations.

Overall, the phenomenon of a horizontal line suggests that, given the specific data and model parameters, changes in dimensions have a relatively minor effect on model performance.

3.3.2 Performance at different K values

Next, I plotted the variation trend of model accuracy, precision, recall rate and F1 value in different dimensions under different K values. This helps to understand the effect of K value on model performance.

```python
import matplotlib.pyplot as plt

# Plotting with matplotlib
fig, axes = plt.subplots(2, 2, figsize=(12, 10))

# Define dimensions
dimensions = [5, 10, 15, 20, 30]

for i, metric in enumerate(metrics):
    ax = axes[i//2, i%2]

    # Plot for each dimension
    for dimension in dimensions:
        dimension_results = results_df[results_df['dimension'] == dimension]
        ax.plot(dimension_results['k'], dimension_results[metric], label=f'Dimension={dimension}')

    ax.set_title(metric.capitalize())
    ax.set_xlabel('K')
    ax.set_ylabel(metric.capitalize())
    ax.legend()

plt.tight_layout()
plt.show()
```

Figure 10: Plot Performance Variation with Different Feature Dimensions

It can be seen from the following group of pictures that the lines of each dimension in the image are overlapped together, which once again proves that the change of dimension has little influence on the change of the model. It can also be seen from this set of graphs that when K is equal to 5, the model shows the best performance in different feature dimensions, including accuracy, precision, recall rate and F1 value.

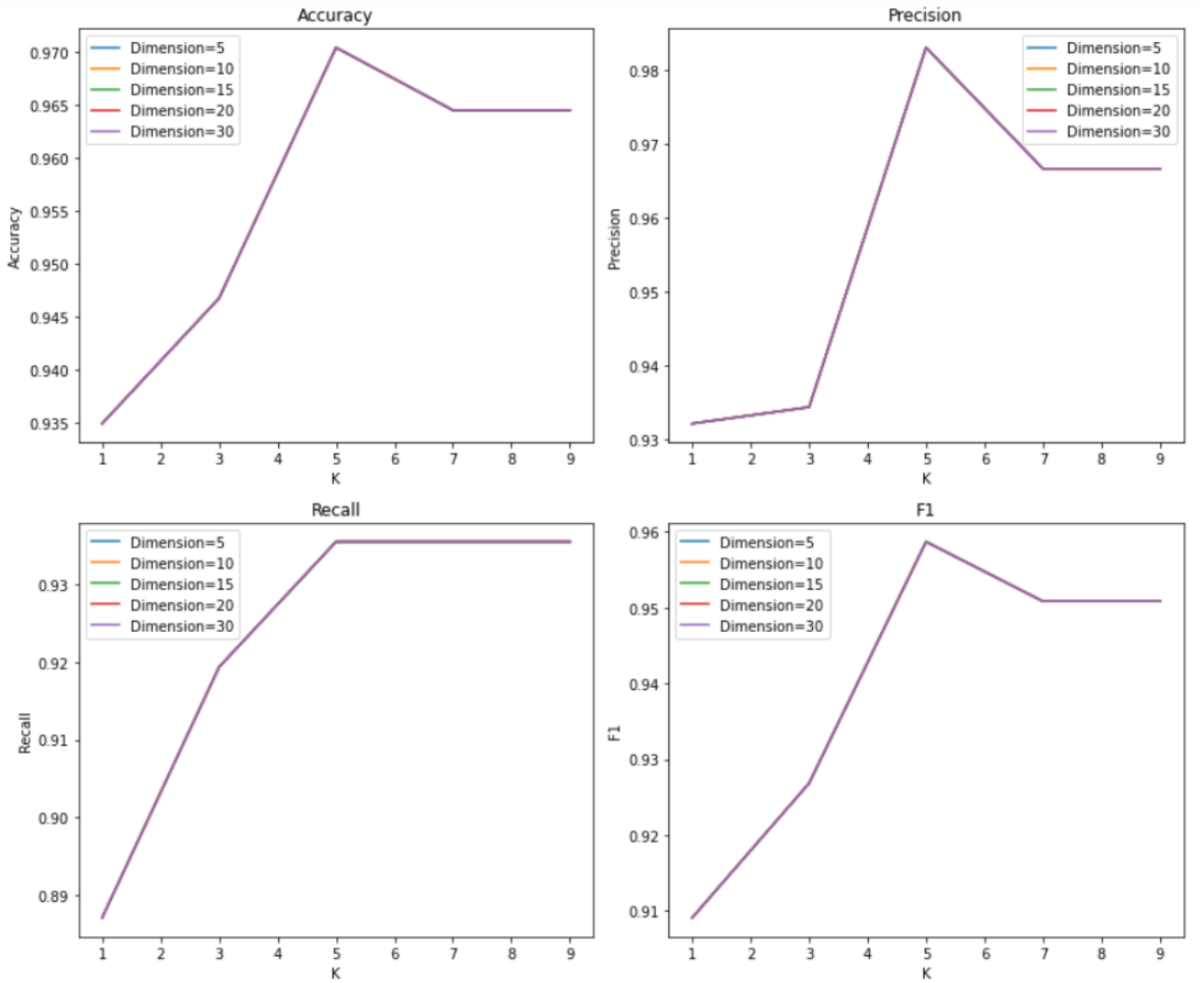

Figure 11: Performance Variation with Different Feature Dimensions

4 Task 3

4.1 MLP Model Design and Performance Evaluation

In Task 3, I design and implement a multi-layer perceptron (MLP) based breast cancer diagnosis system. I used the training data for a 5-fold cross-validation to train and validate different models, and tested various hyperparameters.

4.1.1 The Choice of Feature Dimension

In task 2, I observe that the performance of the model is similar on different dimensions, so I provide two ideas on dimension selection:

- As a first thought, I found that the accuracy of the model improved when the dimensions Ire lower than 5, and the computational cost of the model was relatively low when the dimensions were lower. So with that in mind, I would say 5 is the best choice.
- The second idea is to use dimensions as parameters to train the model and put them in the hyperparameters to choose from.

4.1.2 Hyperparameter Selection

I define a series of combinations of hyperparameters, including the number of hidden layers, the number of hidden nodes, the learning rate, the regularization parameters, the input feature dimension and the decoder. To select the best combination of hyperparameters, I used a 5-fold cross-validation, using average accuracy as the evaluation criterion.

```
# Define the combination of hyperparameters to be tested
layers = [1, 2, 3, 4]
nodes = [50, 100, 150]
learning_rates = [0.001, 0.01, 0.1]
alphas = [0, 0.0001, 0.001, 0.01]
dimensions = [5, 10, 15, 20, 30]
solvers = ['adam']
```

Figure 12: Hyperparameters

4.1.3 Model Training and Evaluation

For each hyperparameter combination, I performed a 50-break cross-validation, using MLPClassifier from the Scikit-learn library, and recorded average accuracy. At the same time, I evaluated the accuracy, precision, recall rate and F1 value of the model on the test data.

```
# Use 5-fold cross-validation to train and validate the model
cross_val_scores = cross_val_score(mlp, X_train_pca, y_train, cv=5, scoring='accuracy')
avg_accuracy = cross_val_scores.mean()

# Record results
results_mlp['layers'].append(layer)
results_mlp['nodes'].append(node)
results_mlp['learning_rate'].append(learning_rate)
results_mlp['alpha'].append(alpha)
results_mlp['dimension'].append(dimension)
results_mlp['solver'].append(solver)
results_mlp['accuracy'].append(avg_accuracy)

# Train the final model (on the entire training set)
mlp.fit(X_train_pca, y_train)

# Evaluate the model on the test set
y_pred = mlp.predict(X_test_pca)
precision = precision_score(y_test, y_pred)
recall = recall_score(y_test, y_pred)
f1 = f1_score(y_test, y_pred)

# Record test set results
results_mlp['precision'].append(precision)
results_mlp['recall'].append(recall)
results_mlp['f1'].append(f1)
```

Figure 13: Model Training and Evaluation

4.2 Interpretation of Result

I organize the results into dataframes and find the combination of hyperparameters that performs best in cross-validation.

```
# Find the index for the best parameter
best_index = results_mlp_df['accuracy'].idxmax()

# Print the best parameters
best_layers = results_mlp_df.loc[best_index, 'layers']
best_nodes = results_mlp_df.loc[best_index, 'nodes']
best_learning_rate = results_mlp_df.loc[best_index, 'learning_rate']
best_alpha = results_mlp_df.loc[best_index, 'alpha']
best_dimension = results_mlp_df.loc[best_index, 'dimension']
best_solver = results_mlp_df.loc[best_index, 'solver']
best_accuracy = results_mlp_df.loc[best_index, 'accuracy']
```

Figure 14: Find the Best Parameter

In Task 3, if the dimension is set to 5 according to the first idea for training, it can be found that when the number of hidden layers is 2, the hidden nodes are 150, the learning rate is 0.01, the regularization parameter is 0.001, and the solver is Adam, the accuracy of the MLP model is 93.75%.

If I use dimension as a parameter and train according to the second idea, I can see that the MLP model can achieve 95% optimal accuracy when the number of hidden layers is 1, the number of hidden nodes is 50, the learning rate is 0.001, the regularization parameter is 0.0, the input feature dimension is 10, and the solver is Adam. This result is obtained through a thorough hyperparameter search and cross-validation.

By comparing the experimental results of the two sets of parameters, I found that the second set of parameters significantly outperformed the first set of parameters in the performance of the MLP model. So I think the second set of parameters is the best parameters for the MLP model. In addition, there is no obvious difference between the two sets of parameters in the performance of the KNN model. Therefore, in the next part of the discussion on model evaluation, I will focus on the KNN and MLP models trained with the second set of parameters to get a full picture of their performance and potential advantages on the problem.

5 Task 4

5.1 Obtain Model Evaluation Data

5.1.1 Train the Best Model

The best models were trained using the optimal parameters identified in the hyperparameter search conducted in Task 3 to assess their performance. The code snippet below demonstrates the process of retraining the models. If the best dimension is not 30, principal component analysis (PCA) is applied to reduce the input features to the optimal dimension. Subsequently, K-nearest neighbors (KNN) and multi-layer perceptron (MLP) models are trained using the transformed data. The KNN model employs 5 neighbors for classification, while the MLP model is configured with the best-hidden layer sizes, learning rate, alpha, and solver obtained from the hyperparameter search. The training time for each model is recorded, and predictions are made on both the test and training datasets. The execution times for KNN and MLP models are then printed for further analysis.

```
from sklearn.neighbors import KNeighborsClassifier
from sklearn.neural_network import MLPClassifier
from sklearn.model_selection import cross_val_score
import time

if best_dimension != 30:
    pca = PCA(n_components=best_dimension)
    X_train_pca = pca.fit_transform(X_train)
    X_test_pca = pca.transform(X_test)
else:
    X_train_pca = X_train
    X_test_pca = X_test

hidden_layer = tuple([best_nodes] * best_layers)

start_time = time.time()
knn_model = KNeighborsClassifier(n_neighbors=5)
knn_model.fit(X_train_pca, y_train)
knn_prediction = knn_model.predict(X_test_pca)
knn_time = time.time() - start_time
knn_train_prediction = knn_model.predict(X_train_pca)

start_time = time.time()
mlp_model = MLPClassifier(hidden_layer_sizes=hidden_layer, learning_rate_init=best_learning_rate, alpha=best_alpha, solver=best_solver)
mlp_model.fit(X_train_pca, y_train)
mlp_prediction = mlp_model.predict(X_test_pca)
mlp_time = time.time() - start_time
mlp_train_prediction = mlp_model.predict(X_train_pca)

print(f'KNN Time: {knn_time}')
print(f'MLP Time: {mlp_time}')
```

Figure 15: Train the Best Model

5.1.2 Obtain Performance Evaluation and Overfitting Analysis Data

The following code snippet evaluates and compares the performance metrics of the K-nearest neighbors (KNN) and multi-layer perceptron (MLP) models on the test dataset, providing insights into their accuracy, precision, recall, F1 score, and overfitting metrics.

```
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score

knn_accuracy = accuracy_score(y_test, knn_prediction)
print(f'KNN Accuracy: {knn_accuracy}')
mlp_accuracy = accuracy_score(y_test, mlp_prediction)
print(f'MLP Accuracy: {mlp_accuracy}')

knn_precision = precision_score(y_test, knn_prediction)
print(f'KNN Precision: {knn_precision}')
mlp_precision = precision_score(y_test, mlp_prediction)
print(f'MLP Precision: {mlp_precision}')

knn_recall = recall_score(y_test, knn_prediction)
print(f'KNN Recall: {knn_recall}')
mlp_recall = recall_score(y_test, mlp_prediction)
print(f'MLP Recall: {mlp_recall}')

knn_f1 = f1_score(y_test, knn_prediction)
print(f'KNN F1: {knn_f1}')
mlp_f1 = f1_score(y_test, mlp_prediction)
print(f'MLP F1: {mlp_f1}')

knn_train_accuracy = accuracy_score(y_train, knn_train_prediction)
knn_test_accuracy = accuracy_score(y_test, knn_prediction)

mlp_train_accuracy = accuracy_score(y_train, mlp_train_prediction)
mlp_test_accuracy = accuracy_score(y_test, mlp_prediction)

knn_overfitting_index = knn_train_accuracy - knn_test_accuracy
mlp_overfitting_index = mlp_train_accuracy - mlp_test_accuracy

print(f'KNN Overfitting Index: {knn_overfitting_index}')
print(f'MLP Overfitting Index: {mlp_overfitting_index}')

knn_overfitting_rate = knn_train_accuracy / knn_test_accuracy
mlp_overfitting_rate = mlp_train_accuracy / mlp_test_accuracy

print(f'KNN Overfitting Rate: {knn_overfitting_rate}')
print(f'MLP Overfitting Rate: {mlp_overfitting_rate}')

knn_overfitting_percentage = ((knn_train_accuracy - knn_test_accuracy) / knn_test_accuracy) * 100
mlp_overfitting_percentage = ((mlp_train_accuracy - mlp_test_accuracy) / mlp_test_accuracy) * 100

print(f'KNN Overfitting Percentage: {knn_overfitting_percentage}%')
print(f'MLP Overfitting Percentage: {mlp_overfitting_percentage}%')
```

Figure 16: Obtain Performance Evaluation and Overfitting Analysis Data

5.2 Comparison of KNN and MLP Models

In Task 2 and Task 3, I designed and implemented a breast cancer diagnosis model based on KNN (K nearest neighbor) and MLP (Multilayer Perceptron) using optimal parameter training, respectively. A comparison of these two models in terms of performance, computational complexity, and overfitting is shown below.

5.2.1 Performance Comparison

Models	KNN	MLP
Accuracy	97.04%	94.08%
Precision	98.31%	100%
Recall	93.55%	83.87%
F1-score	95.87%	91.22%

Table 1: Comparison of Performance Metrics between KNN and MLP

From the table, it can be seen that KNN is higher than MLP in accuracy, recall and f1 scores and lower than MLP in precision.By comparing the performance, overall, the KNN may be the better choice.

5.2.2 Computational Complexity Comparison

Models	KNN	MLP
Computational Complexity	0.00512266	0.03000665

Table 2: Computational Complexity Comparison

In terms of computation time, KNN showed greater efficiency, requiring only about 0.005 seconds compared to about 0.030 seconds for MLP. This shows that KNN is significantly superior to MLP in terms of computational complexity.

5.2.3 Overfitting Comparison

Models	KNN	MLP
Overfitting Index	-0.0479	-0.0333
Overfitting Rate	0.9506	0.9645
Overfitting Percentage	-4.94%	-3.54%

Table 3: Computational Complexity Comparison

In the comparison of overfitting, KNN and MLP showed similar trends. Specifically, both have negative overfit indices of -0.0479 (KNN) and -0.0333 (MLP) respectively, indicating that they perform slightly better on test data than on training data. However, if we observe the overfit rate and overfit percentage, we find that the overfit rate of the KNN model is 0.9506, while the overfit rate of the MLP model is 0.9645. Although the gap between the two is smaller, KNN is slightly better than MLP on this metric. Taking into account the percentage of overfitting, KNN reached -4.94%, while MLP reached -3.54%, further demonstrating KNN's comparative advantage in handling overfitting. Therefore, considering all indicators comprehensively, KNN model performs better in suppressing overfitting and is more reliable than MLP.

6 Discussion

Comprehensively comparing the performance and suitability of KNN and MLP models in breast cancer diagnosis, I derive a series of key observations to guide model selection.

In this data set, I first observe that KNN performs better in terms of Accuracy, which is 97.04% compared to MLP's 94.08%.In terms of Recall, KNN is also significantly ahead at 93.55% compared to MLP's 83.87%.In addition, the F1 scores combining precision and recall are also slightly higher for KNN than MLP. scores are also slightly higher for KNN than for MLP, at 95.87% and 91.22%, respectively, suggesting that the KNN model outperforms the MLP model in terms of the proportion of samples correctly categorized, the percentage of samples predicted to be in the positive category, and the trade-off between precision and recall when considered together. However, it is important to note that the KNN is slightly worse than the MLP in terms of precision,

which is 98.31% for the KNN compared to 100% for the MLP.This may imply that the KNN is relatively poor at capturing positively categorized samples, and may miss some truly positive examples. Although KNN is superior in overall performance, its nonparametric nature classifies through a voting mechanism between neighboring data points, and is better adapted in dealing with irregular decision boundaries and complex data distributions, especially in the case of multiple feature distributions encountered in breast cancer diagnosis.

Secondly, KNN exhibits lower computational complexity, making it suitable for handling high-dimensional data, while MLP is susceptible to the curse of dimensionality. In the breast cancer dataset with high feature dimensions, KNN accomplishes classification through simple distance metrics, whereas MLP requires multiple iterations of complex processes like backpropagation, leading to increased computational complexity. Therefore, in resource-constrained scenarios, KNN proves to be more practical and efficient.

Moreover, in our examination of overfitting, a discernible trend emerges where K-nearest neighbors (KNN) demonstrates a slight superiority over Multilayer Perceptron (MLP) in terms of both overfitting rate and percentage. Specifically, the overfitting rate for KNN stands at 0.9506, outshining MLP's 0.9645. Correspondingly, KNN exhibits an overfitting percentage of -4.94%, whereas MLP registers a slightly higher figure of -3.54%. These comparative statistics suggest that KNN's performance on test data surpasses that on training data, showcasing its relatively superior generalization capability with a lower susceptibility to overfitting. The nuanced advantage of KNN in managing overfitting is attributed to its simplistic model architecture and its emphasis on learning from local patterns. This simplicity results in a more stable performance when faced with overfitting challenges. In contrast, the inherent complexity of MLP, being a deep learning model, bestows it with a robust fitting capacity, but also renders it more prone to overfitting, especially in scenarios characterized by limited data samples. Consequently, this tendency negatively influences the overall generalization performance of the MLP model.

In summary, for breast cancer diagnosis, the KNN model demonstrates superior performance in terms of accuracy, computational complexity, and overfitting. Its high accuracy and relatively lower overfitting make KNN a more reliable choice. While MLP shows competence in certain aspects, its higher computational complexity and slightly inferior performance render it less favorable compared to KNN. Therefore, based on a comprehensive comparison, I recommend choosing the KNN model for breast cancer diagnosis to achieve better results.