COMP 5212 Machine Learning (2024 Spring) Project Report

ZHENG Kaixin
Student ID: 21013165

May 2, 2024

1 Data Processing

Initially, I analyzed each feature and found that they were generally following a normal distribution. Therefore, there was no need to perform normalization or standardization. (In fact, I did try these preprocessing steps and found that they had no significant impact.)

Table 1: Feature Information

Feature	Mean	Std. Dev.	Min	25%	50%	75%	Max
Feature1	-0.4905	1.9286	-6.9058	-1.8006	-0.5001	0.8302	6.4064
Feature2	-0.9880	1.5974	-6.5816	-1.9969	-0.9839	0.0207	5.1490
Feature3	-0.4537	1.9315	-6.8945	-1.7204	-0.4998	0.8019	6.3749
Feature4	0.9763	1.4074	-6.0551	0.0524	1.0204	1.8964	7.6199
Feature5	0.4983	1.9414	-5.9619	-0.8181	0.5096	1.8108	7.1485
Feature6	0.5134	1.8665	-5.8646	-0.7499	0.5185	1.7757	7.2378
Feature7	0.0858	2.0969	-7.0105	-1.3712	0.0263	1.5089	7.4047

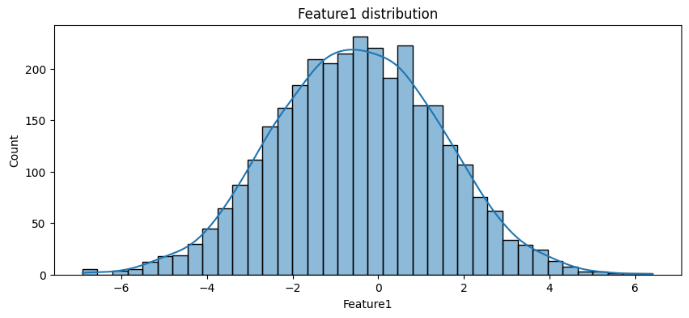

Figure 1: Feature1 distribution

Table 1 shows the descriptive statistics of the features. The distributions of the features do not appear to be significantly different from one another. Figure 1 illustrates the distribution of Feature1, which appears to follow a normal distribution. The other features exhibit similar distributional characteristics.

2 Machine Learning

First, I tested the following models using their default settings, and recorded the resulting accuracy:

Model	Accuracy
LogisticRegression	0.7375
DecisionTreeClassifier	0.805
KNeighborsClassifier	0.9025
SVM	0.91
RandomForestClassifier	0.885
XGBClassifier	0.8875

Based on these results, I decided to use ensemble learning with the four best-performing models, and then tune each individual model using GridSearchCV.

2.1 SVC

-
  ```
  {
    'C': [0.1, 0.5, 1, 10],
    'gamma': ['auto'],
    'kernel': ['rbf', 'poly', 'sigmoid']
  }
  ```

  The best configuration was C=1 and kernel=rbf. This suggests that the polynomial and sigmoid kernels did not perform as well as the rbf kernel.

-
  ```
  {
    'C': [0.1, 0.2...0.9, 1.0],
    'gamma': ['auto'],
    'kernel': ['rbf']
  }
  ```

  Further tuning of the C parameter revealed that C=0.9 was optimal, suggesting a convex relationship between C and performance.

-
  ```
  {
    'C': [0.5, 0.9],
    'gamma': ['scale', 'auto', 1e-3, 1e-4],
    'kernel': ['rbf']
  }
  ```

The final best model was SVC(kernel=”rbf”, C=0.9, gamma=”auto”), which achieved a 5-fold cross-validation score of 0.9119.

2.2 RandomForestClassifier

-
  ```
  {
    'n_estimators': [20, 50, 100],
    'max_depth': [None, 5, 10, 20],
    'min_samples_split': [2, 5, 10],
    'min_samples_leaf': [1, 2, 4]
  }
  ```

  The best hyperparameter were {'max_depth': 20, 'min_samples_leaf': 2, 'min_samples_split': 2, 'n_estimators': 100}. I found that the score increased as the number of estimators and tree depth increased.

-
  ```
  {
    'n_estimators': [100, 200, 500],
    'max_depth': [None, 20, 30],
    'min_samples_split': [2, 5],
    'min_samples_leaf': [1, 2, 4]
  }
  ```

  The best hyperparameter were {'max_depth': 20, 'min_samples_leaf': 1, 'min_samples_split': 2, 'n_estimators': 500}, , though even 500 estimators was not enough.

Testing on the validation set revealed that the accuracy could vary with different random_state values, and that a larger number of estimators did not necessarily lead to better performance. Finally I chose RandomForestClassifier(max_depth=20, min_samples_leaf=2, min_samples_split=2, n_estimators=100, random_state=0) which achieved a score of 0.8819

2.3 KNeighborsClassifier

-
  ```
  {
    'n_neighbors': [3, 5, 7, 10],
    'weights': ['uniform', 'distance'],
    'p': [1, 2]
  }
  ```

  The best configuration was {'n_neighbors': 10, 'p': 2, 'weights': 'distance'}. I considered testing larger values of n and p, but this initial configuration proved to be the optimal.

-
  ```
  {
    'n_neighbors': [10, 15, 20, 25],
    'weights': ['uniform', 'distance'],
    'p': [2, 3, 5]
  }
  ```

  The best one is still {'n_neighbors': 10, 'p': 2, 'weights': 'distance'}.

I selected KNeighborsClassifier(n_neighbors=10, p=2, weights="distance") , which achieved a score of 0.8972.

2.4 XGBClassifier

-
  ```
  {
    'n_estimators': [50, 100, 200],
    'max_depth': [6, 10, 15],
    'learning_rate': [0.01, 0.1, 0.2],
    'gamma': [0, 0.1, 0.2]
  }
  ```

The general trend was that increasing the number of estimators led to better performance.

The best model was XGBClassifier(gamma=0, learning_rate=0.1, max_depth=10, n_estimators=200), which achieved a score of 0.8925.

2.5 VotingClassifier

After tuning the individual models, the accuracy of each was as follows:

Model	Accuracy (validation)	Accuracy (5 fold)
KNeighborsClassifier	0.9175	0.8972
SVM	0.9075	0.9119
RandomForestClassifier	0.9025	0.8819
XGBClassifier	0.9	0.8925

Unfortunately, the voting accuracy of 0.9125 was worse than the best individual model. The model's performance on the test dataset corroborated this. I found that the model was likely overfitting, as the training accuracy was 1.00, but I was unable to resolve this issue.

2.6 Conclusion

Based on the validation dataset results, I chose to use a Voting Classifier ensemble of the KNN and SVM models. The validation accuracy of this approach was 0.92.

3 Deep Learning

3.1 Net Structure

First, I created a large neural network, but the accuracy was under 90%. I was surprised that such a large model was not performing as well as traditional machine learning models. The most likely reason for this is that the model was overfitting.

```
nn.Sequential(
    nn.Linear(7, 128),
    nn.ReLU(),
    nn.Linear(128, 64),
    nn.ReLU(),
    nn.Linear(64, 32),
    nn.ReLU(),
    nn.Linear(32, 1),
    nn.Sigmoid()
)
```

I then used a smaller model, and fortunately the performance improved. However, there was still a gap between the model's performance and that of traditional machine learning models.

```
nn.Sequential(
    nn.Linear(7, 64),
    nn.ReLU(),
    nn.Linear(64, 32),
    nn.ReLU(),
    nn.Linear(32, 16),
    nn.ReLU(),
    nn.Linear(16, 1),
    nn.Sigmoid()
)
```

I believed that using an even larger network could help improve the performance. If a deeper network did not work, I would try a wider network structure. This approach proved successful, as the accuracy easily exceeded 0.9.

```
nn.Sequential(
    nn.Linear(7, 128),
    nn.ReLU(),
    nn.Linear(128, 32),
    nn.ReLU(),
    nn.Linear(32, 1),
    nn.Sigmoid()
)
```

When I attempted to make the network deeper again, the accuracy dropped. After testing several different settings, I settled on a specific network structure.

```
nn.Sequential(
    nn.Linear(7, 256),
    nn.ReLU(),
    nn.Linear(256, 32),
    nn.ReLU(),
    nn.Linear(32, 1),
    nn.Sigmoid()
)
```

3.2 Batch Size

I tested four different batch sizes: [32, 64, 320, 3200]. A batch size of 64 was the best for my network structure, with larger batch sizes resulting in poorer accuracy.

3.3 Epoch

After training the model for each epoch, I calculated the loss on the validation dataset and saved the model with the lowest validation loss. Therefore, running more epochs did not adversely affect the model's performance. In fact, it took 20 to 30 epochs to find the best model.

3.4 Conclusion

n summary, I used the following settings:

- Criterion: BCEWithLogitsLoss()
- Optimizer: Adam(lr=0.01)
- Scheduler: ReduceLROnPlateau('min')
- Epoch: 30
- Batch Size: 64

The net structure is:

```
nn.Sequential(
    nn.Linear(7, 256),
    nn.ReLU(),
    nn.Linear(256, 32),
    nn.ReLU(),
    nn.Linear(32, 1),
    nn.Sigmoid()
)
```

Since the accuracy has already surpassed 0.94, I did not try the other settings.