Practical Assignment – Supervised Machine Learning

(Continuous Assessment One)

By
Prakash Tribhuwan
Student ID - 20034843

Table of Contents

Q1. Data Preparation (What steps would you take to prepare your data? Discuss your approach)

Answer:

1. Loading the Libraries

```
# Loading the libraries
from pandas import read_csv, get_dummies, DataFrame, Series
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split
from imblearn.over_sampling import SMOTE
```

- pandas: It was used for data manipulation and loading the dataset by using (read_csv).
- sklearn: This library contains tools for preprocessing data (StandardScaler), splitting data into training and test sets (train_test_split), and building models.
- imblearn: Includes methods to define class imbalance in datasets (SMOTE).

2. Loading the Dataset:

- read_csv() loads the dataset from a CSV file into a pandas DataFrame. This data set is related to a Bank Data, with customer information.

3. Data exploration and understanding:

```
data.shape    # Dimensions of the dataset
data.info()   # Information about the dataset, including column names, data types, and non-null counts.
data.describe()  # Descriptive statistics for the numerical variables
```

- shape: Displays the dimensions of the dataset (rows, columns).
- info(): This shows metadata about the dataset, including data types and counts of non-null values.
- describe(): Provides basic statistical summaries (mean, std, min, max, etc.) of numerical columns.

Explored the data:

- Checked the size of the dataset with data.shape.
- Viewed basic info like column types and missing values with data.info().
- Got summary statistics (mean, min, max, etc.) using data.describe()

- The target variable y is being checked for value counts. This will help assess class balance (e.g., how many 'yes' vs 'no' entries).

```
data['y'].value_counts()  # Count of target variable values
data['y'].value_counts(normalize=True)  # Proportional count of target variable values
```

4. Data Preprocessing:

- Mapping categorical variables: The default, housing, loan, and y columns, which are categorical with 'yes' and 'no' values, are converted to numeric (1 and 0) to make them usable by machine learning models.

```
data['default'] = data['default'].map({'yes': 1, 'no': 0})  # Convert 'yes'/'no' to 1/0
data['housing'] = data['housing'].map({'yes': 1, 'no': 0})
data['y'] = data['y'].map({'yes': 1, 'no': 0})
data['loan'] = data['loan'].map({'yes': 1, 'no': 0})
```

5. Conversion of categorical features:

- get_dummies(): It turns categorical columns (like job, marital status, education) into separate columns with 0s and 1s. This is important because most machine learning algorithms need numerical data, and this method changes categorical data into a format they can use.
- The resulting dataset now includes new binary columns representing the presence of each category for the listed features, by using data.info() function.

```
data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 4521 entries, 0 to 4520
Data columns (total 17 columns):
 #   Column     Non-Null Count  Dtype
---  ------     --------------  -----
 0   age        4521 non-null   int64
 1   job        4521 non-null   object
 2   marital    4521 non-null   object
 3   education  4521 non-null   object
 4   default    4521 non-null   object
 5   balance    4521 non-null   int64
 6   housing    4521 non-null   object
 7   loan       4521 non-null   object
 8   contact    4521 non-null   object
 9   day        4521 non-null   int64
 10  month      4521 non-null   object
 11  duration   4521 non-null   int64
 12  campaign   4521 non-null   int64
 13  pdays      4521 non-null   int64
 14  previous   4521 non-null   int64
 15  poutcome   4521 non-null   object
 16  y          4521 non-null   object
dtypes: int64(7), object(10)
memory usage: 600.6+ KB
```

```
features = ['job', 'marital', 'education', 'contact', 'month', 'poutcome']
data1 = get_dummies(data, columns=features)
```

6. Feature and target variables:

```
y = data1['y']  # Target variable
X = data1.drop('y', axis=1)  # Features (independent variables)
```

- y: The target variable is 'y', which indicates whether a customer subscribed to the bank's product.
- X: The remaining columns are features used to predict 'y'.

7. Data Standardization:

```
X_scaled = StandardScaler().fit_transform(X)  # Standardize the features
```

- StandardScaler() standardizes the features so that they have a mean of 0 and a standard deviation of 1. This is necessary because many algorithms (like SVM and Random Forest) perform better when the data is on the same scale or same level.

8. Splitting the data into Training and Testing sets:

- train_test_split() splits the data into training and testing sets. Here, 30% of the data is used for testing (test_size=0.3), and the remaining 70% is used for training. The random_state=42 ensures reproducibility.

```
[ ]  X_train, X_test, Y_train, Y_test = train_test_split(X_scaled, y, test_size=0.3, random_state=42)
```

9. Handling Class Imbalance with SMOTE:

SMOTE (Synthetic Minority Over-sampling Technique) is used to balance the class distribution by generating synthetic examples of the minority class. This helps to avoid models being biased toward the majority class, which is common in imbalanced datasets.

```
[ ]  smote = SMOTE(random_state=42)
     X_train, Y_train = smote.fit_resample(X_train, Y_train)  # Over-sample the minority class
```

10. Training the Random Forest Classifier:

```
rf_classifier = ensemble.RandomForestClassifier(n_estimators=50, criterion='entropy', random_state=42)
rf_classifier.fit(X_train, Y_train)
```

- RandomForestClassifier is an ensemble learning method that builds multiple decision trees and collects their predictions.
- n_estimators=50: The number of trees to grow.
- criterion='entropy': The criterion used to measure the quality of a split (in this case, information gain).
- random_state=42 ensures the results are reproducible.

```
Y_pred = rf_classifier.predict(X_test)  # Predict on the test set
```

- The model is then used to predict the target variable Y_pred for the test set.

11. Evaluating the Random Forest Model:

```
# Evaluating the random forest
from sklearn import metrics
Accuracy= metrics.accuracy_score(Y_test,Y_pred) # Calculating the accuracy of the model build
Recall= metrics.recall_score(Y_test,Y_pred) # Calculating the recall of the model build
Precision= metrics.precision_score(Y_test,Y_pred) # Calculating the precision of the model build
print('Accuracy:', Accuracy)
print('Recall:', Recall)
print('Precision:', Precision)
```

```
Accuracy: 0.8894620486366986
Recall: 0.3092105263157895
Precision: 0.5108695652173914
```

Q2. Model Hyperparameter Tuning (Which hyperparameters would you tune and why? How would you tune them?)

Answer:

To find the best settings for these hyperparameters, you should use Grid Search or Random Search along with cross-validation to carefully test different combinations.

(i) Random Forest Classifier (RFC)

➢ Hyperparameters to Tune:

- n_estimators: This defines the number of trees in the forest. Increasing the number of trees generally improves the performance but increases processing or run time.
- max_features: This controls how many features the model looks at when splitting a node. By limiting the number of features, we can help prevent overfitting.
- Tuning Strategy: We used GridSearchCV to find the optimal n_estimators value, specifically testing values between 50 and 350. The tuning was done based on the precision score as we want to reduce false positives.

```
from sklearn.model_selection import GridSearchCV

# Selecting the optimal n_estimators parameter
RF1=ensemble.RandomForestClassifier(n_estimators=50,criterion='entropy',max_features=None,random_state=42)
ntrees={'n_estimators': [50,100,150,180,220,250,300,320,350]} # Grid of n_estimators parameter length = 8

grid_search=GridSearchCV(estimator=RF1,param_grid=ntrees,scoring='precision',cv=5)

grid_search.fit(X_train,Y_train) # Fitting decision tree classifier to the training set
best_nestimator= grid_search.best_params_ # Selecting the optimal parameter n_estimators
print(best_nestimator)
```

```
{'n_estimators': 100}
```

(ii) Support Vector Classifier (SVC)

➢ Hyperparameters to Tune:

- kernel: The kernel is a function that changes the data into a higher-dimensional space. Common types are linear, rbf, poly, and sigmoid.

- C: This is the regularization parameter that helps balance margin maximization and classification errors. A higher value of C increases model complexity.

- Tuning Strategy: We used GridSearchCV with different values for kernel and C, focusing on maximizing recall to ensure the minority class is predicted effectively.

Q3. Choice of Evaluation Metric (Which metric would be suitable for model evaluation and why?)

Suitable Metrics for Model Evaluation:

Choosing the right evaluation metric depends on the problem, especially when working with imbalanced data, like in our case where the target variable shows whether a customer subscribes to a service or not.

1. Accuracy
The proportion of correct predictions (both true positives and true negatives) to the total number of predictions made.

Formula:

$$\textbf{Accuracy} = \frac{\textbf{True Positives + True Negatives}}{\textbf{Total Samples}}$$

Accuracy works well when the dataset is balanced, meaning the classes are about equal. But for imbalanced datasets (like ours, where most of the values are 0 and few are 1), accuracy can be misleading. A model that always predicts the majority class could still have high accuracy, even if it struggles to detect the minority class.

In our dataset: The accuracy is calculated, but given the imbalanced nature of our dataset (with more customers not subscribing), accuracy might not fully reflect the model's performance. It could give you a false sense of how well your model is performing.

2. Precision

The proportion of positive predictions that are actually correct. Precision answers the question, "Out of all the instances classified as positive, how many were actually positive?"

Formula:

$$\textbf{Precision} = \frac{\textbf{True Positives}}{\textbf{True Positives + False Positives}}$$

Precision is important when the cost of false positives is high. In this context, a false positive means predicting a customer will subscribe when they actually won't. If we want to minimize unnecessary marketing efforts or misallocating resources to customers who are unlikely to subscribe, precision becomes crucial.

In our dataset: We calculate precision, which is helpful for determining how well your model is at correctly identifying subscribers (positives). This is important in business scenarios where we want to avoid offering services to customers who are unlikely to subscribe.

3. Recall

The proportion of actual positives that were correctly identified. Recall answers the question, "Out of all the actual positive instances, how many did I correctly identify?"

Formula:

$$\textbf{Recall} = \frac{\textbf{True Positives}}{\textbf{True Positives + False Negatives}}$$

When to Use:

Recall is important when the cost of missing positive instances is high. In our case, a false negative means a customer who would have subscribed was missed by the model. If the business goal is to identify as many subscribers as possible, even at the cost of having some false positives, recall is the better metric to focus on.

In our dataset: Recall is also calculated, and it is very important in business applications where we want to ensure that as many subscribers as possible are identified, even if some false positives occur.

4. F1-Score

The harmonic mean of precision and recall. The F1-score balances the trade-off between precision and recall.

Formula:

$$F1 = 2 \times \frac{Precision \times Recall}{Precision + Recall}$$

The F1-score is especially useful when we need a balance between precision and recall. It is particularly important in scenarios where both false positives and false negatives have serious consequences (e.g., marketing resource allocation, where both misidentifying potential subscribers and missing actual subscribers can be costly).

In our dataset: The F1-score is calculated, which is a good metric to evaluate the overall performance of the model, especially in imbalanced datasets like this. If we care about both identifying as many subscribers as possible (recall) and not wasting resources on false positives (precision), the F1-score will give a balanced measure.

5. Confusion Matrix:

The confusion matrix provides a detailed breakdown of the model's predictions, showing the true positives (TP), false positives (FP), true negatives (TN), and false negatives (FN). This matrix is essential for calculating other metrics like precision, recall, and F1-score.

The confusion matrix looks like this:

	Predicted Positive (1)	Predicted Negative (0)
Actual Positive (1)	True Positive (TP)	False Negative (FN)
Actual Negative (0)	False Positive (FP)	True Negative (TN)

Why Use Confusion Matrix?

- The confusion matrix helps to better understand the types of errors the model is making, which can guide further improvement.
- It also allows us to calculate precision, recall, and F1-score directly, which are more informative for imbalanced datasets.

NOTE : Confusion Matrix isn't used as we have used Important Features/Validation which is much efficient as it eliminates the irrelevant columns from the dataset.

- feature_importance : _ gives the importance of each feature in the prediction. This is useful for feature selection or understanding which features are most impactful in predicting the target variable. Attached a screenshot for reference.

```
# Selecting the optimal features
rf_classifier = ensemble.RandomForestClassifier(n_estimators=100, criterion='entropy', random_state=42)
rf_classifier.fit(X_train, Y_train)
feature_importances = rf_classifier.feature_importances_
imp_features = Series(feature_importances, index=X_train.columns).sort_values(ascending=False)
print(imp_features)
```

Recommended Metric would be:

A combination of Precision and Recall (to capture as many subscribers as possible), and F1-Score (to balance precision and recall) are likely the best metrics for this task because the dataset is imbalanced.

```
Accuracy: 0.8305084745762712
Precision: 0.37337662337662336
Recall: 0.756578947368421
F1 score: 0.5
```

```
Accuracy: 0.8533529845246868
Precision: 0.3842364532019704
Recall: 0.5131578947368421
F1 score: 0.4394366197183099
```

Q4. Overfitting avoidance mechanism (Which mechanism (feature Selection/regularization) would you use and why?)

Answer:

Overfitting happens when a machine learning model learns patterns that aren't important, like random noise in the training data, instead of focusing on the real patterns. This makes the model do well on the training data but badly on new data (test data). To prevent overfitting, two common methods are used: Feature Selection and Regularization. Both of these help make the model simpler, but they do it in different ways.

- For Random Forest, the use of regularization via max_features and n_estimators helps in controlling the difficulty of individual trees and ensures that the model doesn't overfit to the training data. Further hyperparameter tuning (like optimizing max_depth and n_estimators) would help find the best regularization balance.

Importance of Feature Selection:

```
feature_importances = rf_classifier.feature_importances_
imp_features = Series(feature_importances, index=X_train.columns).sort_values(ascending=False)
print(imp_features)
```

Regularization:

```
RF = ensemble.RandomForestClassifier(
    n_estimators=50,
    criterion='entropy',
    random_state=42
)
```

- For SVC, the C parameter helps control the balance between making the model fit the training data perfectly and keeping the decision boundary smooth. GridSearchCV is used to find the best value for this parameter and the kernel, making sure the model works well on new data.

```
ker_c = {'classification__C': [0.001, 0.01, 0.1, 1, 10]}
grid_search1 = GridSearchCV(estimator=model, param_grid=ker_c, scoring='recall', cv=5)

svm_classifier_rbf = SVC(kernel='poly', C=10, gamma=0.1, random_state=42)  # Regularizing with poly kernel
```

In summary, GridSearchCV combined with proper regularization parameters (max_depth, max_features, C, etc.) is the most effective method for avoiding overfitting in your models, as it ensures the best model parameters are chosen based on cross-validation performance.

- For Random Forest, focusing on the number of estimators and tree depth helps prevent overfitting.
- For SVC, tuning the C value, selecting an appropriate kernel, and controlling the poly parameter are key to managing overfitting.

These mechanisms, especially with cross-validation, allow the models to achieve a balance between bias and variance, thereby avoiding overfitting.

Hence, in summary, Regularization (by controlling the depth and minimum samples for split and leaf) would be the primary mechanism for avoiding overfitting in Random Forest.

In both models, Feature Selection can also help reduce overfitting by removing irrelevant or redundant features, but it is a secondary mechanism compared to regularization in these specific models.

Q5. Results analysis a). Which of the two models (random forest or support vector classifier) would you recommend for deployment in the real-world? b). Is any model underfitting? If yes, what could be the possible reasons?

Answer:

(a)

Random Forest is fast, flexible, and easy to use. It works well with large and messy data, makes accurate predictions, and doesn't need much fine-tuning. This makes it a better choice for real-world problems where data can be unpredictable and complex.

1. Works well with Big Data:

Random Forest can handle large datasets easily and doesn't take too long to train. It's good at working with lots of information, which is common in real-world problems.

2. Stays accurate even with messy data:

Random Forest is strong against mistakes (overfitting) and works well even if the data is unbalanced or noisy. This is important because real-world data is often messy or not perfectly organized.

3. Performs well:

Random Forest performs strongly in predicting both positive and negative outcomes (good at both precision and recall). It's also faster compared to other models like Support Vector Classifier (SVC), especially when data is complex.

4. Easier to understand:

Random Forest is a bit more complex than some models, but it can tell you which features (like columns or data points) are most important for making decisions. This makes it easier to understand compared to models like SVC, which can be tricky, especially with complicated settings.

5. Fewer settings to tune:

Random Forest is easier to set up and doesn't need as much fine-tuning. You don't have to spend as much time adjusting the model, which makes it quicker and simpler to use in real-world scenarios.

Conclusion: Deploy Random Forest for most real-world scenarios where scalability, robustness, and ease of deployment are priorities. It is well-suited for datasets with large feature spaces, imbalanced classes, and complex interactions or relation between features.

(b) Underfitting occurs when a model is too simple or not complex enough to capture the patterns in the data. This leads to poor performance on both the training set and test set. A model that is underfitting will have low accuracy, precision, recall, and other evaluation metrics on the training data, as well as on unseen test data.

Which in our case is not true, none of the metrics is underfit, both our sets are performing well, there is a relationship observed between the features, hence there is no underfitting observed in our dataset.