Comparative Analysis of Random Forest Classifier and Support Vector Machine for Bank Dataset.

Introduction:

This report examines the effectiveness of two widely-used machine learning models, Random Forest Classifier (RFC) and Support Vector Machine (SVM), in a classification problem. It utilizes a bank marketing dataset to predict whether customers will subscribe to a term deposit. The report outlines the process of preparing the data, implementing the algorithms, and evaluating their performance through various metrics.

Data Preparation:

The Dataset was prepared as follows:

To start with, I have loaded required libraries. After that the dataset was imported from a CSV file using the Pandas library.

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

```
[2] my_dataset=read_csv('/content/bank.csv') #reading the working csv file
```

For categorical feature encoding:

a. The columns `default`, `housing`, `loan` and `y` (which is also our target class) were converted to numerical values by mapping to 1 (yes) and to 0 (No).

```
[5] #categorical feature encoding using map function
    # converting categorical data to numerical data

    my_dataset['default']=my_dataset['default'].map({'yes':1,'no':0})
    my_dataset['housing']=my_dataset['housing'].map({'yes':1,'no':0})
    my_dataset['loan']=my_dataset['loan'].map({'yes':1,'no':0})
    my_dataset['y']=my_dataset['y'].map({'yes':1,'no':0})
```

b. The columns `job`, `marital`, `education`, `contact`, `month` and `poutcome` were transformed into binary variables via one-hot encoding with the function. And **"`data10`"** is our new dataset.

```
# now we need to use get dummies function to all objects (Data type) - this will covert all unique rows
# we will not use get dummies function to int64 and float data type as int is already in numeric format
data10=get_dummies(my_dataset,['job','marital','education','contact','month','poutcome'],dtype=int)
```

The dependent variable `(y)`, which indicates subscription status, was defined as `y`, while all other columns were defined as `x` which are independent variables.

```
y=data10['y']   #"y" is dependent variable
x=data10.drop('y',axis=1) # we will drop "y" targets value and define x as independent(to all other column)
print(y.shape)
print(x.shape)
```

```
(4521,)
(4521, 48)
```

Data Scaling

Data scaling was performed using the `x_scaled=StandardScaler().fit_transform(x)` function. This step is crucial for addressing differences in feature scales, which can enhance the performance and stability of machine learning algorithms like Random Forest and SVM that were implemented later.

```
x_scaled=StandardScaler().fit_transform(x)
DataFrame(x_scaled)
```

When the entire dataset is scaled together, identical parameters are applied consistently to both the training and test sets. This method preserves the data's relative structure while avoiding potential bias that could result from scaling the sets independently.

Data Splitting

Data splitting was performed using `train_test_split(x_scaled, y, test_size=0.30, random_state=40).`" This line describes the process of dividing the dataset into two distinct parts: a training set (70%) and a testing set (30%).

The core reason for this split is to prevent bias in evaluating the model's performance. If we trained and tested on the same data, the model might simply memorize the data, leading to an overly optimistic (and misleading) assessment of its ability to handle new, unseen data.

```
x_train,x_test,y_train,y_test=train_test_split(x_scaled,y, test_size=0.30, random_state=40)
print(x_train.shape)
print(x_test.shape)
print(y_train.shape)
print(y_test.shape)
```

The training set `(x_train, y_train)` is used to teach models to recognize patterns. The testing set `(x_test, y_test)` is held back from the training process and is used to assess how well the trained model generalizes to new, unseen data. By making predictions on the test set `(y_pred1, y_pred2)` and comparing these predictions to the actual values in `y_test`, we can estimate how our model would perform on real-world data it has never encountered before.

Simply put, splitting data enables us to train a model on one portion of the dataset and then evaluate its performance on a different, independent portion, providing a realistic measure of its effectiveness.

Data Balancing

Data balancing was performed on the training set `(x_train, y_train)` using SMOTE (Synthetic Minority Over-sampling Technique) after splitting the dataset. The goal of this step was to address the class imbalance in the target variable, `y`.

```
[ ]  x_train,y_train =SMOTE (random_state = 40).fit_resample(x_train,y_train)
     y_train.value_counts()
```

y	count
1	2811
0	2811

dtype: int64

The `y_train.value_counts()` output after applying SMOTE confirmed that both classes **(1 and 0)** in the training set were equalized **(2811 instances each**). This indicates that the original training data had an uneven class distribution, a common issue in classification problems.

In imbalanced datasets, machine learning models often lean towards the majority class, achieving high overall accuracy but underperforming on the minority class. Balancing the dataset helps prevent this bias. By applying SMOTE to the training data aimed to enhance the performance of both the Random Forest and SVM classifiers.

This will end our Data Preparation task and we will start with our first model **Random Forest Classifier**

A Random Forest Classifier is an ensemble learning algorithm that constructs multiple decision trees during the training process. For classification tasks, the model predicts the class that is selected by the majority of the trees.

Each tree is trained on a random subset of the training data using bootstrapping. When a tree splits a node, only a random subset of features is considered for the split, which increases model diversity. This randomness reduces overfitting and enhances the overall robustness and accuracy of the model compared to individual decision trees.

Workflow for the RFC Model

1. **Importing Necessary Libraries:**
   The `RandomForestClassifier` was imported from `sklearn.ensemble`, along with evaluation metrics from `sklearn.metrics`.
2. **Initial Model Building and Training:**
   - A `RandomForestClassifier` object named `RF_classifier11` was instantiated with the following hyperparameters:
     - `n_estimators=50` (number of trees in the forest)
     - `criterion='entropy'` (function used to measure the quality of a split)
     - `random_state=40` (to ensure reproducibility).
   - The model was trained on the balanced training data (`x_train, y_train`) using the `fit()` method.
3. **Making Predictions:**
   - The trained `RF_classifier11` was used to make predictions on the test set (`x_test`) with the `predict()` method.
4. **Evaluating the Initial Model:**
   - The model's **accuracy** was computed by comparing the predicted values (`y_pred1`) with the actual values in the test set (`y_test`) using `metrics.accuracy_score()`.
   - **Recall** was calculated using `metrics.recall_score()`.
   - **Precision** was determined using `metrics.precision_score()`.
   - The results were printed to assess the initial model's performance.

```
#from sklearn.ensemble import RandomForestClassifier
from sklearn import ensemble
from sklearn import metrics
RF_classifier11 =ensemble.RandomForestClassifier(n_estimators=50, criterion='entropy', random_state=40)  # building model
RF_classifier11.fit(x_train,y_train)#training
y_pred1=RF_classifier11.predict(x_test)# testing

accuracy = metrics.accuracy_score(y_test, y_pred1)  # calculating accuracy
print("Accuracy: ", round(accuracy, 2))

recall = metrics.recall_score(y_test, y_pred1)
print("Recal: ", round(recall, 2))

precision = metrics.precision_score(y_test, y_pred1)
print("Precision: ", round(precision, 2))
```

```
Accuracy:  0.88
Recal:  0.32
Precision:  0.52
```

5. **Hyperparameter Tuning with GridSearchCV:**
    - Another `RandomForestClassifier`, named `RF_classifier2`, was instantiated with `criterion='entropy'` and `random_state=40`.
    - A grid of hyperparameters (`no_trees`) was defined, including `n_estimators` (number of trees) and `max_features` (number of features to consider for the best split).
    - **GridSearchCV** was applied to perform an exhaustive search across the hyperparameter grid.
    - The training and evaluation process for different hyperparameter combinations was conducted using the entire scaled dataset `(x_scaled, y)`
    - The best hyperparameters (`{'max_features': 'sqrt', 'n_estimators': 65}`) were identified and printed using `grid_search1.best_params_.`

```
from sklearn.model_selection import GridSearchCV
RF_classifier2 = ensemble.RandomForestClassifier(criterion='entropy', random_state=40) # building model
no_trees = {'n_estimators': [50,60,65,70,77,98,106],
            'max_features': ['sqrt', 'log2']}
grid_search1 = GridSearchCV(estimator=RF_classifier2, param_grid=no_trees, scoring='recall', cv=4)
grid_search1.fit(x_scaled, y)## training, testing , evaluation, ranking.
best_parameters = grid_search1.best_params_
print(best_parameters)
```

```
{'max_features': 'sqrt', 'n_estimators': 65}
```

6. **Building and Training the Optimized Model:**
    - A new `RandomForestClassifier` object, also named `RF_classifier11`, was created using the best hyperparameters (`n_estimators=65`, `criterion='entropy'`, `max_features='sqrt'`, and `random_state=40`).
    - This model was trained on the balanced training data (`x_train, y_train`).
7. **Feature Importance Analysis:**
    - The importance of each feature in the trained `RF_classifier11` was computed using `feature_importances_`.
    - A pandas Series was created to organize the importance scores, and the features were sorted in descending order.
    - The results highlighted the relative contribution of each feature to the model's predictions.

```
#from sklearn.ensemble import RandomForestClassifier
from sklearn import ensemble
import pandas as pd
from sklearn import metrics

RF_classifier11 =ensemble.RandomForestClassifier(n_estimators=65, criterion='entropy', max_features='sqrt', random_state=40)
RF_classifier11.fit(x_train,y_train) # fitting random forest to training data
y_pred1=RF_classifier11.predict(x_test)# testing

imp_features = pd.Series(RF_classifier11.feature_importances_, index=list(x)).sort_values(ascending=False)
print(imp_features)

accuracy_RF = metrics.accuracy_score(y_test, y_pred1)  # calculating accuracy
print("Accuracy: ", accuracy_RF)

recall_RF = metrics.recall_score(y_test, y_pred1)
print("Recall:", recall_RF)

precision_RF = metrics.precision_score(y_test, y_pred1)
print("Precision:", precision_RF)
```

8. **Evaluating the Optimized Model:**
    - Predictions were made on the test set (`x_test`) using the retrained `RF_classifier11`.
    - Metrics such as **accuracy**, **recall**, and **precision** were calculated and printed to evaluate the model's performance with the tuned hyperparameters.
9. **Model with Important Features:**
    - The top 80% of the most important features were selected based on the feature importance analysis.
    - New training and testing sets (`x_train_top, x_test_top`) were created using only these selected features.
    - A `RandomForestClassifier` named `RF_classifier12` was instantiated using the same best hyperparameters.
    - This model was trained using the reduced feature set (`x_train_top, y_train`) and evaluated on the test set (`x_test_top`).
    - The performance metrics (accuracy, recall, precision) were compared to the initial model.

```
num_top_features = int(len(imp_features) * 0.8)
top_features = imp_features.index[:num_top_features]

# Convert x_train and x_test to DataFrames if they are NumPy arrays
x_train = pd.DataFrame(x_train, columns=x.columns)
x_test = pd.DataFrame(x_test, columns=x.columns)

x_train_top = x_train[top_features]
x_test_top = x_test[top_features]

RF_classifier12 = ensemble.RandomForestClassifier(n_estimators=65, criterion='entropy', max_features='sqrt', random_state=40)
RF_classifier12.fit(x_train_top, y_train)
y_pred2 = RF_classifier12.predict(x_test_top)

print("Original Model:")
print("Accuracy:", round(metrics.accuracy_score(y_test, y_pred1), 2))
print("Recall:", round(metrics.recall_score(y_test, y_pred1), 2))
print("Precision:", round(metrics.precision_score(y_test, y_pred1), 2))

print("\nModel with important feature:")
print("Accuracy:", round(metrics.accuracy_score(y_test, y_pred2), 2))
print("Recall:", round(metrics.recall_score(y_test, y_pred2), 2))
print("Precision:", round(metrics.precision_score(y_test, y_pred2), 2))
```

```
Original Model:
Accuracy: 0.88
Recall: 0.34
Precision: 0.53

Model with important feature:
Accuracy: 0.88
Recall: 0.39
Precision: 0.53
```

Support Vector Machine (SVM)

Process:

1. **Initial Model Construction:** A preliminary SVM classifier (`SV_classifier1`) is created using the training dataset (`x_train, y_train`) with the `svm.SVC(random_state=40)` function. Its purpose is to determine a hyperplane that separates the classes effectively.
2. **Prediction Phase:** The initial classifier predicts outcomes (`Y_pred1`) using the test dataset (`x_test`).
3. **Performance Evaluation:** The model's performance is assessed using metrics such as accuracy, recall, and precision.
4. **Hyperparameter Optimization with GridSearchCV:**
    - A pipeline (`SVM_classifier2`) is developed, incorporating SMOTE for class balancing followed by an `SVC` classifier.
    - A grid (`kernels_c`) of potential **kernels** (`'linear', 'poly', 'rbf', 'sigmoid'`) and various **regularization parameters (C)** `(.001, .01, 0.1, 0.5, 1, 2)` is created.
    - The `GridSearchCV` function evaluates all combinations of these parameters using a 4-fold cross-validation on the training dataset. The focus is on maximizing the **recall** metric.
    - The optimal parameters (`best_parameters`) and the best recall result (`best_result`) are identified.
5. **Optimized Model Development:** A final SVM classifier (`SV_classifier3`) is built, utilizing the **best kernel (`'rbf'`)** and **optimal C value** discovered through the grid search.
6. **Final Evaluation:** This optimized model undergoes training on the complete training dataset, and its performance is tested using the test data.

```
# Now we will use SVM model for our prediction
from sklearn import svm
SV_classifier1 = svm.SVC(random_state = 40) # building SVM classifier
SV_classifier1.fit(x_train, y_train) # training
Y_pred1= SV_classifier1.predict(x_test)  #testing
```

```
from sklearn import metrics
Accuracy=metrics.accuracy_score(y_test, y_pred1) # Calculating accuracy
print("Accuracy: ", round(Accuracy, 2))

recall = metrics.recall_score(y_test, y_pred1) # Calculate recall
print ("Recall: ", round(recall,2))

precision = metrics.precision_score(y_test, y_pred1) # Calculate precision
print ("Precision: ", round(precision,2))
```

```
Accuracy:  0.88
Recall:  0.34
Precision:  0.53
```

Reason for Selecting the Parameters:

The parameters selected for `GridSearchCV`—including a variety of kernels and C values—allow for a thorough evaluation of how these hyperparameters influence the SVM's performance and ability to generalize to new, unseen data.

1. **Kernel Exploration:** Incorporating diverse kernels (such as `'linear'` for linear separability, and `'poly', 'rbf'`, and `'sigmoid'` for non-linear relationships) ensures that the grid search identifies the most suitable kernel for achieving the best recall on your dataset, especially when the optimal kernel isn't known in advance.
2. **Regularization Strength (C) Exploration:** The `C` parameter balances the trade-off between achieving a larger margin and minimizing training errors:
    - Lower values of `C` (e.g., `.001, .01, 0.1`) favor larger margins, possibly resulting in underfitting.
    - Higher values of `C` (e.g., `0.5, 1, 2`) reduce training errors, which might lead to overfitting. By testing a broad range of `C` values, `GridSearchCV` identifies the optimal balance that maximizes recall while ensuring the model generalizes effectively.

```
from imblearn.pipeline import Pipeline
from sklearn.svm import SVC
SVM_classifier2 = Pipeline([('balancing', SMOTE(random_state = 40)),('classification', SVC(random_state = 40) ) ]) # bulilding SVM classifier with pipeline and SVC

kernels_c = {'classification__kernel': ['linear','poly','rbf','sigmoid'], 'classification__C': [.001,0.01,0.1,0.5,1,2]} # Defining the parameter grid for GridSearchCV.
grid_search1 = GridSearchCV(estimator=SVM_classifier2, param_grid=kernels_c, scoring='recall', cv=4) #applying GridSearchCV
grid_search1.fit(x_train,y_train)

best_parameters = grid_search1.best_params_ #getting the best parameters from GridSearchCV.
print(best_parameters)
best_result = grid_search1.best_score_ #Getting the best score
print(best_result)
```

```
{'classification__C': 2, 'classification__kernel': 'rbf'}
0.970832168200589 2
```

Hyperparameter Tuning for Random Forest and SVM Models

Random Forest Classifier:

- **Tuned Hyperparameters:**
    - `n_estimators`: Determines the number of trees in the forest, balancing prediction stability with computational cost. Grid search tested values ranging from 50 to 106.
    - `max_features`: Defines the number of features considered for each split, promoting diversity among trees and reducing overfitting. Tested options included `'sqrt'` and `'log2'`.
- **Tuning Process:**
  A grid search performed 4-fold cross-validation using `GridSearchCV`. The recall score was prioritized to identify the best configuration. The optimal parameters were `n_estimators: 65` and `max_features: 'sqrt'`.

Support Vector Machine (SVM):

- **Tuned Hyperparameters:**
  - kernel: Specifies the hyperplane type for data separation. Explored kernel options were `'linear'`, `'poly'`, `'rbf'`, and `'sigmoid'`.
  - `C`: The regularization parameter controls the trade-off between margin width and classification accuracy. Values from `0.001` to `2` were tested.
- **Tuning Process:**
  A pipeline incorporating SMOTE addressed class imbalance, followed by `GridSearchCV` testing kernel and `C` combinations via 4-fold cross-validation. Recall was optimized, with the best setup being a `rbf` kernel and `C: 2`.

Summary:
Both models employed `GridSearchCV` to systematically explore hyperparameters, optimizing recall as the scoring metric to enhance performance and generalization.

Why you selected all the parameters in the grid search:

I have selected the range of kernels (`'linear', 'poly', 'rbf', 'sigmoid'`) and the range of `C` values (`.001, .01, 0.1, 2, 20, 200`) in the `GridSearchCV` to systematically explore how these hyperparameters affect the SVM's performance (specifically, the recall score in your case).

- **Exploring Different Kernels:** Different kernels allow the SVM to learn different types of decision boundaries. By including multiple kernels in the grid search, you allowed the process to automatically determine which type of boundary (linear, polynomial, radial, or sigmoid) is most suitable for your data to achieve the best recall. You didn't know beforehand which kernel would perform best, so you included a variety of common and powerful options.
- **Exploring Different Regularization Strengths (`C`):** The `C` parameter controls the trade-off between fitting the training data well and having a generalizable model.
  - By including very small values of `C` (like `0.001` and `0.01`), you tested models that prioritize a large margin, potentially at the cost of some training errors. This can help prevent overfitting if your data has noise or outliers.
  - By including intermediate values of `C` (like `0.1` and `2`), you tested models with a balance between margin maximization and error minimization.
  - By including larger values of `C` (like `20` and `200`), you tested models that prioritize minimizing training errors, potentially leading to a smaller margin and a risk of overfitting.

  The provided code mitigates overfitting through two primary strategies: feature selection and regularization. Feature selection is explicitly implemented by using a Random Forest Classifier to rank feature importance, subsequently retaining only the top 80% for model training. This process eliminates redundant or irrelevant features that could introduce noise, thereby simplifying the model and improving its generalization capabilities. Additionally, this approach enhances interpretability and reduces computational demands.

Regularization, both directly and indirectly, is also employed. In the Random Forest model, the `max_features` parameter functions as an indirect form of regularization by limiting the number of features considered during tree splits, thus controlling complexity. For the Support Vector Machine (SVM), the C parameter serves as a direct regularization mechanism, balancing model fit and generalization. Smaller C values favor larger margins and prevent overfitting, while larger values risk overfitting by overly complex decision boundaries. GridSearchCV is used to find the optimal C value. Feature selection is ideal when dealing with datasets containing many irrelevant features, when model interpretability is crucial, or when computational costs need to be minimized. Regularization is more appropriate when most features are useful and the aim is to control overall model complexity. Often, combining both strategies yields the best performance. By integrating feature selection and regularization, the developers have effectively addressed the risk of overfitting, ensuring robust model performance on both training and unseen data.

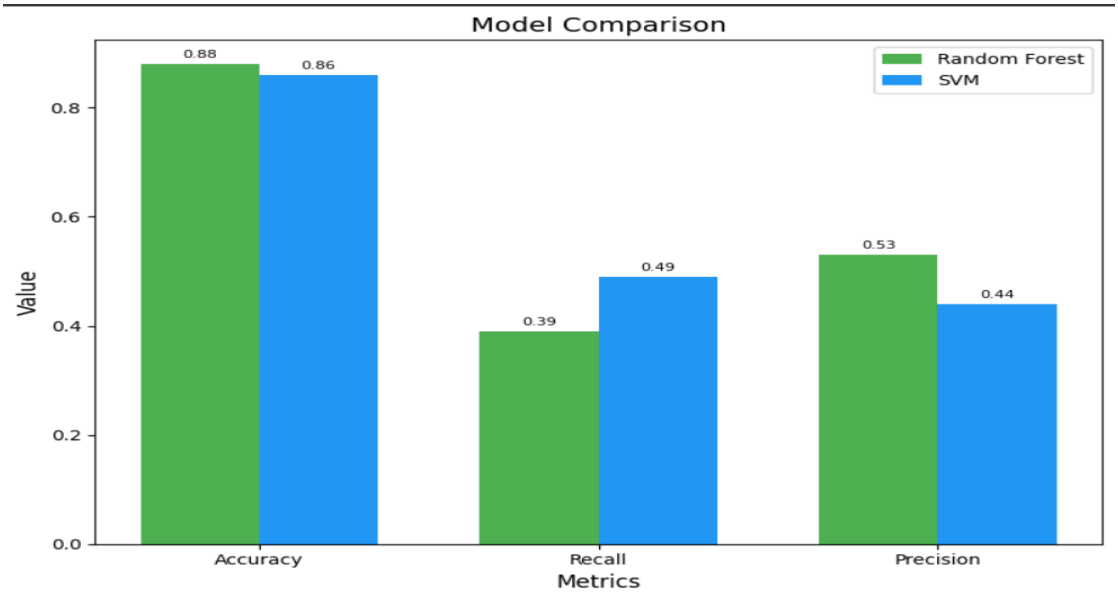
Conclusion

The Model for Real Time Deployment

I have compared the best-performing, post-tuning versions of each model.

- **Random Forest:** Trained on the top 80% of features (RF_classifier12), the Random Forest model exhibited these approximate test dataset results: Accuracy: 0.88, Recall: 0.39, Precision: 0.53.
- **Support Vector Machine (SVM):** The optimally tuned SVM model (SV_classifier3) demonstrated these test dataset results: Accuracy: 0.86, Recall: 0.49, Precision: 0.44.

Comparative metrics show Random Forest has slightly better accuracy (**0.88** vs. 0.86), while SVM has higher recall (**0.49** vs. 0.39), indicating improved identification of potential subscribers. Random Forest has higher precision (**0.53** vs. 0.44).



Model selection depends on the relative importance of recall and precision. If missing potential subscribers is costly, **SVM** is preferred for its higher recall, despite lower accuracy and precision. If expending resources on unlikely subscribers is costly, Random Forest is preferred for its higher precision.

Given that recall was the primary tuning metric, **SVM** might be favored where maximizing potential positive case identification is paramount.