Minsoo Lee

**Individual Assignment 5**

**Experiment**

Our Train & Test data : [Cell Phone Churn - TRAIN-1.csv](https://canvas.asu.edu/courses/197217/files/90002491/download?wrap=1) , [Cell Phone Churn - TEST-1.csv](https://canvas.asu.edu/courses/197217/files/90002170/download?wrap=1)

Results using default parameters

A black and white screen with numbers

Description automatically generated

Every classifier’s result is very high.

After doing the RandomizedSearchCV, the best parameter for Random Forest is

{'n\_estimators': 100, 'min\_samples\_split': 20, 'min\_samples\_leaf': 2, 'max\_depth': 40,, 'bootstrap': False}

And the following result is below:



In this experiment, hyperparameter tuning did not result in any improvement over the default Random Forest classifier for any metric. This outcome highlights that:

1. For datasets with clear patterns and strong features, the default configuration of Random Forest can often perform exceptionally well.
2. Hyperparameter tuning is most beneficial when the dataset is noisy, complex, or imbalanced, or when the default performance is suboptimal.

**Feature importances**

**A black text on a white background

Description automatically generated**

**Result for Stacking KNN:**



The results show that **stacking with KNN** did not improve upon any of the individual classifiers in terms of accuracy, precision, recall, or F1 score. All metrics remain identical across all models, including the stacked model.

**Conclusion**

**Things that we learned**

1) Dataset Simplicity: The dataset is straightforward, with highly predictive features, leading to near-perfect performance across all models.

2) Minimal Impact of Tuning: Hyperparameter tuning and stacking did not improve results as the default models already performed optimally.

3) Redundant Predictions: The base models produced similar predictions, limiting the effectiveness of stacking.

**Ideas to Try for Better Results**

1) Feature Engineering: Create new features or transformations to add complexity.

2) Introduce Noise: Add noise or remove dominant features to test model robustness.

3) Change Stacking Classifier: Use Logistic Regression or Gradient Boosting as the stacking classifier.

4) Evaluate Generalization: Test on a separate dataset or use cross-validation to ensure robustness.

Python Code

import pandas as pd

import numpy as np

from sklearn.model\_selection import train\_test\_split

from sklearn.preprocessing import LabelEncoder, StandardScaler, OneHotEncoder

from sklearn.impute import SimpleImputer

from sklearn.pipeline import Pipeline

from sklearn.compose import ColumnTransformer

from sklearn.metrics import accuracy\_score, precision\_score, recall\_score, f1\_score

from sklearn.neural\_network import MLPClassifier

from sklearn.svm import LinearSVC

from sklearn.tree import DecisionTreeClassifier

from sklearn.ensemble import RandomForestClassifier

from sklearn.neighbors import KNeighborsClassifier

# Load data

train\_path = '/content/drive/MyDrive/Colab Notebooks/CIS508\_Machine\_Learning/Individual\_Assignment/Cell Phone Churn - TRAIN-1.csv'

test\_path = '/content/drive/MyDrive/Colab Notebooks/CIS508\_Machine\_Learning/Individual\_Assignment/Cell Phone Churn - TEST-1.csv'

train\_df = pd.read\_csv(train\_path)

test\_df = pd.read\_csv(test\_path)

# Remove columns with all missing values (like 'CONVB')

train\_df = train\_df.dropna(axis=1, how='all')

test\_df = test\_df.dropna(axis=1, how='all')

# Features and target

X\_train = train\_df.drop(columns=['TARGET'])

y\_train = train\_df['TARGET']

X\_test = test\_df.drop(columns=['TARGET'])

y\_test = test\_df['TARGET']

# Define categorical and numerical columns

categorical\_cols = ['SEX', 'SPORTS', 'ARTS', 'TRAVEL', 'EDUC']

numeric\_cols = [col for col in X\_train.columns if col not in categorical\_cols]

# Ensure numeric columns contain only numeric data

for col in numeric\_cols:

X\_train[col] = pd.to\_numeric(X\_train[col], errors='coerce')

X\_test[col] = pd.to\_numeric(X\_test[col], errors='coerce')

# Preprocessing pipelines

numeric\_transformer = Pipeline(steps=[

('imputer', SimpleImputer(strategy='median')), # Handle missing numeric values

('scaler', StandardScaler())]) # Standardize numeric data

categorical\_transformer = Pipeline(steps=[

('imputer', SimpleImputer(strategy='most\_frequent')), # Handle missing categorical values

('onehot', OneHotEncoder(handle\_unknown='ignore'))]) # One-hot encode categorical data

# Combine preprocessors

preprocessor = ColumnTransformer(

transformers=[

('num', numeric\_transformer, numeric\_cols),

('cat', categorical\_transformer, categorical\_cols)])

# Apply preprocessing

X\_train\_preprocessed = preprocessor.fit\_transform(X\_train)

X\_test\_preprocessed = preprocessor.transform(X\_test)

# Model 1: MLPClassifier

print("Training MLPClassifier with default settings...")

mlp = MLPClassifier(random\_state=42) # Default settings

mlp.fit(X\_train\_preprocessed, y\_train)

y\_pred\_mlp = mlp.predict(X\_test\_preprocessed)

# Metrics for MLPClassifier

accuracy\_mlp = accuracy\_score(y\_test, y\_pred\_mlp)

precision\_mlp = precision\_score(y\_test, y\_pred\_mlp, pos\_label=1)

recall\_mlp = recall\_score(y\_test, y\_pred\_mlp, pos\_label=1)

f1\_mlp = f1\_score(y\_test, y\_pred\_mlp, pos\_label=1)

print(f"MLPClassifier Metrics:\nAccuracy: {accuracy\_mlp:.2f}, Precision: {precision\_mlp:.2f}, Recall: {recall\_mlp:.2f}, F1 Score: {f1\_mlp:.2f}")

# Model 2: LinearSVC

print("\nTraining LinearSVC with default settings...")

svc = LinearSVC(random\_state=42) # Default settings

svc.fit(X\_train\_preprocessed, y\_train)

y\_pred\_svc = svc.predict(X\_test\_preprocessed)

# Metrics for LinearSVC

accuracy\_svc = accuracy\_score(y\_test, y\_pred\_svc)

precision\_svc = precision\_score(y\_test, y\_pred\_svc, pos\_label=1)

recall\_svc = recall\_score(y\_test, y\_pred\_svc, pos\_label=1)

f1\_svc = f1\_score(y\_test, y\_pred\_svc, pos\_label=1)

print(f"LinearSVC Metrics:\nAccuracy: {accuracy\_svc:.2f}, Precision: {precision\_svc:.2f}, Recall: {recall\_svc:.2f}, F1 Score: {f1\_svc:.2f}")

# Model 3: DecisionTreeClassifier

print("\nTraining DecisionTreeClassifier with default settings...")

tree = DecisionTreeClassifier(random\_state=42) # Default settings

tree.fit(X\_train\_preprocessed, y\_train)

y\_pred\_tree = tree.predict(X\_test\_preprocessed)

# Metrics for DecisionTreeClassifier

accuracy\_tree = accuracy\_score(y\_test, y\_pred\_tree)

precision\_tree = precision\_score(y\_test, y\_pred\_tree, pos\_label=1)

recall\_tree = recall\_score(y\_test, y\_pred\_tree, pos\_label=1)

f1\_tree = f1\_score(y\_test, y\_pred\_tree, pos\_label=1)

print(f"DecisionTreeClassifier Metrics:\nAccuracy: {accuracy\_tree:.2f}, Precision: {precision\_tree:.2f}, Recall: {recall\_tree:.2f}, F1 Score: {f1\_tree:.2f}")

# Model 4: RandomForestClassifier

print("\nTraining RandomForestClassifier with default settings...")

forest = RandomForestClassifier(random\_state=42) # Default settings

forest.fit(X\_train\_preprocessed, y\_train)

y\_pred\_forest = forest.predict(X\_test\_preprocessed)

# Metrics for RandomForestClassifier

accuracy\_forest = accuracy\_score(y\_test, y\_pred\_forest)

precision\_forest = precision\_score(y\_test, y\_pred\_forest, pos\_label=1)

recall\_forest = recall\_score(y\_test, y\_pred\_forest, pos\_label=1)

f1\_forest = f1\_score(y\_test, y\_pred\_forest, pos\_label=1)

print(f"RandomForestClassifier Metrics:\nAccuracy: {accuracy\_forest:.2f}, Precision: {precision\_forest:.2f}, Recall: {recall\_forest:.2f}, F1 Score: {f1\_forest:.2f}")

from sklearn.model\_selection import RandomizedSearchCV

# Define the Random Forest classifier

rf = RandomForestClassifier(random\_state=42)

# Hyperparameter space

param\_distributions = {

'n\_estimators': [50, 100, 200, 300, 400], # Number of trees

'max\_depth': [None, 10, 20, 30, 40, 50], # Maximum depth of the tree

'min\_samples\_split': [2, 5, 10, 20], # Minimum samples required to split a node

'min\_samples\_leaf': [1, 2, 4, 6], # Minimum samples required at a leaf node

'bootstrap': [True, False] # Whether bootstrap samples are used

}

# Randomized search

random\_search = RandomizedSearchCV(

estimator=rf,

param\_distributions=param\_distributions,

n\_iter=50, # Number of random parameter settings

cv=3, # 3-fold cross-validation

scoring='f1', # Use F1 score as the metric

random\_state=42,

n\_jobs=-1 # Use all processors

)

# Perform random search

print("Starting RandomizedSearchCV...")

random\_search.fit(X\_train\_preprocessed, y\_train)

# Best parameters and best estimator

print(f"Best parameters: {random\_search.best\_params\_}")

best\_rf = random\_search.best\_estimator\_

# Evaluate on the test set

y\_pred\_best\_rf = best\_rf.predict(X\_test\_preprocessed)

accuracy\_rf = accuracy\_score(y\_test, y\_pred\_best\_rf)

precision\_rf = precision\_score(y\_test, y\_pred\_best\_rf, pos\_label=1)

recall\_rf = recall\_score(y\_test, y\_pred\_best\_rf, pos\_label=1)

f1\_rf = f1\_score(y\_test, y\_pred\_best\_rf, pos\_label=1)

print("\nBest Random Forest Classifier Metrics:")

print(f"Accuracy: {accuracy\_rf:.2f}, Precision: {precision\_rf:.2f}, Recall: {recall\_rf:.2f}, F1 Score: {f1\_rf:.2f}")

# Truncate feature names to match preprocessed dataset length

all\_feature\_names = all\_feature\_names[:X\_train\_preprocessed.shape[1]]

# Ensure the lengths match before proceeding

if len(feature\_importances) != len(all\_feature\_names):

raise ValueError("Mismatch persists between feature names and importances even after truncation.")

# Create a DataFrame for feature importances

feature\_importances\_df = pd.DataFrame({

'Feature': all\_feature\_names,

'Importance': feature\_importances

})

# Sort by importance and extract the top 5 features

top\_features = feature\_importances\_df.sort\_values(by='Importance', ascending=False).head(5)

# Display the top 5 features

print("\nTop 5 Features by Importance:")

print(top\_features)

# Step 1: Train base classifiers

mlp = MLPClassifier(random\_state=42)

svc = LinearSVC(random\_state=42)

tree = DecisionTreeClassifier(random\_state=42)

forest = RandomForestClassifier(random\_state=42)

# Fit base classifiers

print("Training base classifiers...")

mlp.fit(X\_train\_preprocessed, y\_train)

svc.fit(X\_train\_preprocessed, y\_train)

tree.fit(X\_train\_preprocessed, y\_train)

forest.fit(X\_train\_preprocessed, y\_train)

# Step 2: Generate predictions as input for stacking

print("Generating predictions for stacking...")

train\_predictions = np.column\_stack([

mlp.predict(X\_train\_preprocessed),

svc.predict(X\_train\_preprocessed),

tree.predict(X\_train\_preprocessed),

forest.predict(X\_train\_preprocessed)

])

test\_predictions = np.column\_stack([

mlp.predict(X\_test\_preprocessed),

svc.predict(X\_test\_preprocessed),

tree.predict(X\_test\_preprocessed),

forest.predict(X\_test\_preprocessed)

])

# Step 3: Train the stacking classifier (KNN)

print("Training KNN as stacking classifier...")

knn = KNeighborsClassifier()

knn.fit(train\_predictions, y\_train)

# Step 4: Evaluate the stacking model

y\_pred\_ensemble = knn.predict(test\_predictions)

accuracy\_ensemble = accuracy\_score(y\_test, y\_pred\_ensemble)

precision\_ensemble = precision\_score(y\_test, y\_pred\_ensemble, pos\_label=1)

recall\_ensemble = recall\_score(y\_test, y\_pred\_ensemble, pos\_label=1)

f1\_ensemble = f1\_score(y\_test, y\_pred\_ensemble, pos\_label=1)

# Display results

print("\nEnsemble Stacking (KNN) Metrics:")

print(f"Accuracy: {accuracy\_ensemble:.2f}")

print(f"Precision: {precision\_ensemble:.2f}")

print(f"Recall: {recall\_ensemble:.2f}")

print(f"F1 Score: {f1\_ensemble:.2f}")