Final Code

December 6, 2023

[ ]:
```
# Import necessary libraries for data visualization
import matplotlib.pyplot as plt  # Matplotlib for plotting
import seaborn as sns  # Seaborn for statistical data visualization

# Import machine learning libraries
from sklearn.ensemble import GradientBoostingRegressor  # Gradient Boosting␣
↪Regressor
import pandas as pd  # Pandas for data manipulation
import numpy as np  # NumPy for numerical operations
```

[ ]:
```
 # Read the dataset into a Pandas DataFrame
df = pd.read_csv('C:/Users/rashi/OneDrive/Documents/4th sem/Capstone Project/
↪Final Dataset.csv')
```

[ ]:
```
# Display the first few rows of the DataFrame to inspect the data
df.head()
```

[ ]:
```
# Provide information about the DataFrame, including data types and missing␣
↪values
df.info()
```

[ ]:
```
# Remove non-numeric characters from the "Weighted salary" column and convert␣
↪it to float
df["Weighted salary"] = df["Weighted salary"].str.replace('[^\d.]', '',␣
↪regex=True).astype(float)
```

[ ]:
```
# Check the number of missing values in each column
df.isnull().sum()
```

[ ]:
```
# Define a list of columns to impute with the median
columns_to_impute = ["GMAT/GRE Required"]
```

[ ]:
```
# Impute missing values in selected columns with the median
for column in columns_to_impute:
    median_value = df[column].median()
    df[column].fillna(median_value, inplace=True)
```

[ ]:
```
# Define a threshold for the maximum number of missing values allowed
threshold = 250  # Adjust this threshold based on your dataset
```

[ ]:
```
# Drop columns with a high number of missing values
columns_to_drop = df.columns[df.isnull().sum() > threshold]
df.drop(columns=columns_to_drop, inplace=True)
```

[ ]:
```
# Check the number of missing values in each column
df.isnull().sum()
```

[ ]:
```
# Generate descriptive statistics for the dataset
df.describe()
```

[ ]:
```
# Setting the style for the plots
sns.set_style("whitegrid")

# Create a new figure
plt.figure(figsize=(14, 6))

# Create a subplot of 1 row and 2 columns for box plot and histogram
plt.subplot(1, 2, 1)
sns.boxplot(y=df['Weighted salary'])
plt.title('Box plot of Weighted salary')

plt.subplot(1, 2, 2)
sns.histplot(df['Weighted salary'], kde=True, bins=30)
plt.title('Histogram of Weighted salary')

# Setting the style for the plots
sns.set_style("whitegrid")

# Create a new figure
plt.figure(figsize=(14, 6))

# Create a subplot of 1 row and 2 columns for box plot and histogram
plt.subplot(1, 2, 1)
sns.boxplot(y=df['International Diversity'])
plt.title('Box plot of International Diversity')

plt.subplot(1, 2, 2)
sns.histplot(df['International Diversity'], kde=True, bins=30)
plt.title('Histogram of International Diversity')

# Setting the style for the plots
sns.set_style("whitegrid")

# Create a new figure
```

```
plt.figure(figsize=(14, 6))

# Create a subplot of 1 row and 2 columns for box plot and histogram
plt.subplot(1, 2, 1)
sns.boxplot(y=df['Salary percentage increase'])
plt.title('Box plot of Salary percentage increase')

plt.subplot(1, 2, 2)
sns.histplot(df['Salary percentage increase'], kde=True, bins=30)
plt.title('Histogram of Salary percentage increase')

# Setting the style for the plots
sns.set_style("whitegrid")

# Create a new figure
plt.figure(figsize=(14, 6))

# Create a subplot of 1 row and 2 columns for box plot and histogram
plt.subplot(1, 2, 1)
sns.boxplot(y=df['Faculty with doctorates (%)'])
plt.title('Box plot of Faculty with doctorates (%)')

plt.subplot(1, 2, 2)
sns.histplot(df['Faculty with doctorates (%)'], kde=True, bins=30)
plt.title('Histogram of Faculty with doctorates (%)')

# Setting the style for the plots
sns.set_style("whitegrid")

# Create a new figure
plt.figure(figsize=(14, 6))

# Create a subplot of 1 row and 2 columns for box plot and histogram
plt.subplot(1, 2, 1)
sns.boxplot(y=df['Effective Emp rate'])
plt.title('Box plot of Effective Emp rate')

plt.subplot(1, 2, 2)
sns.histplot(df['Effective Emp rate'], kde=True, bins=30)
plt.title('Histogram of Effective Emp rate')

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

[ ]:

```
# Calculate feature correlations
correlation_matrix = df.corr()
correlations_with_target = correlation_matrix["#"].sort_values(ascending=False)
```

```
print(correlations_with_target)
```

[ ]:
```
# Multivariate Analysis (Correlation Heatmap)
corr_matrix = df.corr()
plt.figure(figsize=(25, 15))
sns.heatmap(corr_matrix, annot=True, cmap='coolwarm')
plt.title('Correlation Heatmap')
plt.show()
```

[ ]:
```
import pandas as pd
from sklearn.ensemble import RandomForestRegressor
import xgboost as xgb
import matplotlib.pyplot as plt

# Replace the target variable and feature columns as needed
target_column = "#"
feature_columns = [ "Weighted salary"
,"Salary percentage increase"
,"Value for money rank"
,"Career progress rank"
,"Aims achieved (%)"
,"Effective Emp rate"
,"International work mobility rank"
,"International course experience rank"
,"Faculty with doctorates (%)"
,"Internship"
,"Overall Satisfaction"
,"International Diversity"
,"GMAT/GRE Required"
,"Female Empowerment Score"
]

X = df[feature_columns]  # Features
y = df[target_column]    # Target variable


# Method 1: Random Forest
rf_model = RandomForestRegressor()
rf_model.fit(X, y)
feature_importance_rf = rf_model.feature_importances_

# Method 2: XGBoost
xgb_model = xgb.XGBRegressor()
xgb_model.fit(X, y)
feature_importance_xgb = xgb_model.feature_importances_

# Display feature importance results for each method
```

```
print("Feature Importance (Random Forest):", feature_importance_rf)
print("Feature Importance (XGBoost):", feature_importance_xgb)

# Visualize feature importances using bar plots

plt.figure(figsize=(12, 6))
plt.bar(range(len(feature_importance_rf)), feature_importance_rf)
plt.xlabel('Features')
plt.ylabel('Feature Importance Score (Random Forest)')
plt.title('Feature Importance Scores (Random Forest)')
plt.xticks(range(len(feature_importance_rf)), feature_columns, rotation=90)
plt.show()

plt.figure(figsize=(12, 6))
plt.bar(range(len(feature_importance_xgb)), feature_importance_xgb)
plt.xlabel('Features')
plt.ylabel('Feature Importance Score (XGBoost)')
plt.title('Feature Importance Scores (XGBoost)')
plt.xticks(range(len(feature_importance_xgb)), feature_columns, rotation=90)
plt.show()
```

[ ]:

```
metrics = [ "Weighted salary"
,"Salary percentage increase"
,"Value for money rank"
,"Career progress rank"
#,"Career service rank"
,"Aims achieved (%)"
#,"Employed at three months (main)"
#,"Employed at three months (additional)"
,"Effective Emp rate"
#,"Female faculty (%)"
,"Female students (%)"
#,"Women on board (%)"
#,"International faculty (%)"
#,"International students (%)"
#,"International board (%)"
,"International work mobility rank"
,"International course experience rank"
,"Faculty with doctorates (%)"
#,"Three year average"
,"Internship"
#,"Overall Satisfaction"
#,"No. of Students per Staff"
,"International Diversity"
#,"GMAT/GRE Required"
#,"Female Empowerment Score"
]
```

```python
from sklearn.datasets import make_regression
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestRegressor
from sklearn.svm import SVR
from sklearn.metrics import mean_squared_error, mean_absolute_error, r2_score
import numpy as np

X= df[metrics]
y= df["#"]

# 2. Split data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,␣
↪random_state=42)

# 3. Train Random Forest and SVM models
rf = RandomForestRegressor(n_estimators=100, random_state=42)
rf.fit(X_train, y_train)

svm = SVR(kernel='linear')
svm.fit(X_train, y_train)

# 4. Predict on test data
y_pred_rf = rf.predict(X_test)
y_pred_svm = svm.predict(X_test)

# 5. Calculate and print metrics
metrics = {
    "MSE": mean_squared_error,
    "MAE": mean_absolute_error,
    "RMSE": lambda y_true, y_pred: np.sqrt(mean_squared_error(y_true, y_pred)),
    "R^2": r2_score
}

print("Random Forest Metrics:")
for name, func in metrics.items():
    print(f"{name}: {func(y_test, y_pred_rf)}")

print("\nSVM Metrics:")
for name, func in metrics.items():
    print(f"{name}: {func(y_test, y_pred_svm)}")
```

[ ]:

```python
# Assuming you have already defined your X and y as features and target variable
# If not, use X = df[metrics] and y = df["#"] as you did in your previous code

metrics = [ "Weighted salary"
,"Salary percentage increase"
```

```
,"Value for money rank"
,"Career progress rank"
,"Aims achieved (%)"
,"Effective Emp rate"
,"International work mobility rank"
,"International course experience rank"
,"Faculty with doctorates (%)"
,"Internship"
,"Overall Satisfaction"
,"International Diversity"
,"GMAT/GRE Required"
,"Female Empowerment Score"
]

from sklearn.datasets import make_regression
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestRegressor
from sklearn.svm import SVR
from sklearn.metrics import mean_squared_error, mean_absolute_error, r2_score
import numpy as np

X= df[metrics]
y= df["#"]

# 2. Split data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,␣
↪random_state=42)

# 3. Create and train the Gradient Boosting Regressor model
gb_regressor = GradientBoostingRegressor(n_estimators=100, random_state=42)
gb_regressor.fit(X_train, y_train)

# 4. Predict on the test data
y_pred_gb = gb_regressor.predict(X_test)

# 5. Calculate and print regression metrics
metrics = {
    "MSE": mean_squared_error,
    "MAE": mean_absolute_error,
    "RMSE": lambda y_true, y_pred: np.sqrt(mean_squared_error(y_true, y_pred)),
    "R^2": r2_score
}

print("Gradient Boosting Metrics:")
for name, func in metrics.items():
    print(f"{name}: {func(y_test, y_pred_gb)}")
```

[ ]:
```python
from sklearn.model_selection import RandomizedSearchCV

# Simplified hyperparameter grid
simple_param_grid = {
    'learning_rate': [0.01, 0.1, 0.3],
    'n_estimators': [50, 100, 200],
    'max_depth': [3, 5, 7],
    'min_child_weight': [1, 3],
    'gamma': [0, 0.2],
    'subsample': [0.8, 1.0],
    'colsample_bytree': [0.8, 1.0]
}

# Randomized search with cross-validation (fewer iterations for simplicity)
xgb_simple_random_search = RandomizedSearchCV(xgb.XGBRegressor(objective ='reg:
 ↪squarederror', random_state=42),
                                              ␣
 ↪param_distributions=simple_param_grid,
                                              n_iter=20,
                                              scoring='neg_mean_squared_error',
                                              cv=3,
                                              verbose=1,
                                              n_jobs=-1,
                                              random_state=42)

xgb_simple_random_search.fit(X_train, y_train)

# Best hyperparameters from the simplified search
simple_best_params = xgb_simple_random_search.best_params_
simple_best_params
```

[ ]:
```python
import xgboost as xgb
from sklearn.metrics import mean_squared_error, r2_score
from sklearn.metrics import mean_absolute_error

# Given training and testing sets: X_train, X_test, y_train, y_test

# Train XGBoost with the optimal hyperparameters
optimal_xgb = xgb.XGBRegressor(
    subsample=0.8,
    n_estimators=200,
    min_child_weight=3,
    max_depth=3,
    learning_rate=0.1,
    gamma=0.2,
    colsample_bytree=0.8,
    objective ='reg:squarederror',
```

```python
    random_state=42
)

optimal_xgb.fit(X_train, y_train)

# Predict on the test set
y_pred_optimal = optimal_xgb.predict(X_test)

# Evaluate the model's performance
mse_optimal = mean_squared_error(y_test, y_pred_optimal)
r2_optimal = r2_score(y_test, y_pred_optimal)
mae_optimal = mean_absolute_error(y_test, y_pred_optimal)
rmse_optimal = np.sqrt(mse_optimal)

print(f"MSE: {mse_optimal}")
print(f"R^2: {r2_optimal}")
print(f"MAE: {mae_optimal}")
print(f"RMSE: {rmse_optimal}")
```

[ ]:

```python

```