# Machine Learning Assignment 2

| Name | ID | Task |
|---|---|---|
| **Malak Alaa Mohamed** | 23012115 | Dataset Selection,Data Splitting,K-Nearest Neighbors (KNN) Algorithm |
| **Maram Mohamed Mahmoud** | 23012197 | Cross-Validation,Confusion Matrix: |
| **Ganna Mohamed Abdul Fattah** | 23012138 | Overfitting and Model Improvement,Visualization |

# K-Nearest Neighbors (KNN)

## Introduction:

**Dataset:** https://www.kaggle.com/datasets/larsen0966/student-performance-data-set

We used the student-por.csv dataset, which contains student characteristics (school, age, study time, absences, etc.) and grades (G1, G2, G3).

Our goal: Predict whether a student passes or fails based on these features, using a K-Nearest Neighbors (KNN) classifier.
We defined "pass" as G3 ≥ 10, "fail" as G3 < 10.

## Data Preprocessing:

### Step 1: Load and Explore the Dataset

### Step 2: Define the Target Variable (Y) and Features (X)

1. We created a new binary column called pass:
    - 1 for pass and 0 for fail

```
# Create the 'pass' target column
# If the final grade G3 is 10 or more then pass = 1 otherwise pass=0
df['pass'] = df['G3'].apply(lambda grade: 1 if grade >= 10 else 0)
```

2. Feature Selection

- We removed G1, G2, G3 from features to prevent "cheating" (future data leaking into training)
- This problem is called "Data Leakage": Data leakage happens when your model accidentally learns from information it would not have during real prediction.
  It leads to artificially high validation/test scores, but bad performance in real life.

```
# Drop G1, G2, and G3 (grades) from features if you want to prevent leaking the answer
features = df.drop(columns=['G1', 'G2', 'G3', 'pass'])
target = df['pass']
```

## 3. Encoding

- KNN needs numerical features because it calculates distances
  So, we converted categorical features (like sex, school, address) into numbers using one-hot encoding (dummy variables).

```
# Convert categorical variables into numeric variables
features_encoded = pd.get_dummies(features, drop_first=True)
```

## 4. Splitting the Data

We split the dataset into three parts

- Training set (60%) — to train the model.
- Validation set (20%) — to tune and find the best K value.
- Test set (20%) — to evaluate the final model.
- We used stratified splitting to keep the ratio of pass/fail the same across splits.

```
# split into 60% training (to teach the KNN model)
# 40% temporary
X_train, X_temp, y_train, y_temp = train_test_split(features_encoded, target, test_size=0.4, random_state=42, stratify=target)
```

```
# 50% validation, 50% test from temp (20% each)
# 20% validation  To choose the best K
# test 20%  To test the final model
# stratify=target (Keeps pass/fail balance)

X_val, X_test, y_val, y_test = train_test_split(X_temp, y_temp, test_size=0.5, random_state=42, stratify=y_temp)
```

## KNN Model:

1. Import tools:

- One to build a KNN model (KNeighborsClassifier).
- One to check how good your predictions are (accuracy_score).

```
from sklearn.neighbors import KNeighborsClassifier #to build the KNN model
from sklearn.metrics import accuracy_score #to check the accuracy of the model
```

2. List of K values to try:

- We tried different values of K: 1, 3, 5, 7, 9, 11, 13, and 15.
- For each K, we trained the model using the training set and checked its accuracy on the validation set.

```
# Try different values of K
#Odd numbers are better to avoid ties in voting
k_values = [1, 3, 5, 7, 9, 11, 13, 15]

# for ever k we try check the accuracy and save it into the list then we can compare to find the best
validation_accuracies = []
```

3. Try Different K Values:

For each value of K:

- Create the KNN model with that k.
- Train it using the training data (fit).
- Predict the validation set (predict).
- Calculate how accurate the prediction was.
- Save the accuracy to compare later

```
for k in k_values:
    # Create KNN model
    knn = KNeighborsClassifier(n_neighbors=k)

    # Fit on training data
    knn.fit(X_train, y_train)

    # Predict on validation data
    y_val_pred = knn.predict(X_val)

    # Calculate accuracy
    accuracy = accuracy_score(y_val, y_val_pred)
    #save that accuracy
    validation_accuracies.append(accuracy)
    print(f"K = {k}: Validation Accuracy = {accuracy:.4f}")
```

4. Pick the K with the best accuracy:
    - The k with the best accuracy is K=11 with accuracy= 0.8538

```
K = 1: Validation Accuracy = 0.7538
K = 3: Validation Accuracy = 0.8231
K = 5: Validation Accuracy = 0.8308
K = 7: Validation Accuracy = 0.8308
K = 9: Validation Accuracy = 0.8462
K = 11: Validation Accuracy = 0.8538
K = 13: Validation Accuracy = 0.8385
K = 15: Validation Accuracy = 0.8385
```

## Cross validation:

- We used cross-validation to improve the reliability of our model's predictions and to tune hyperparameters. In our KNN model, the key hyperparameter is K (the number of neighbors). Cross-validation helps us:
- Find the best K value for the highest accuracy.
- Prevent overfitting by training the model on different subsets of the data

## It works like the following :

- We test different K values and evaluate each using k-fold cross-validation.
- The data is split into k folds—each fold takes a turn as the test set while the rest are used for training.
- After running all folds, we average the accuracy scores to get a final estimate.

## Before starting we needed to know :

- Define K values – Possible numbers of neighbors to test (already prepared).
- Split the data – Training, validation, and test sets (already done).
- Choose k-folds – Number of folds for cross-validation (5 or 10). We choose 5
- Report average accuracy – Compare results across all folds.
- Compare scores – Check cross-validation accuracy vs. validation set vs. test set results.

```
kf = KFold(n_splits=5, shuffle=True , random_state=42)
cv_accuracies = []
```

we made a list cv_accuracies to append the avg acc from each combination = k_value

using a loop that iterates over each combination and in each one it applies cv and the output is an array with all acc for the model over these folds then we get the avg and append it to the list after finishing all combinations we get the best k which gives the highest acc from the list

```
kf = KFold(n_splits=5, shuffle=True , random_state=42)
cv_accuracies = []

for k in k_values:
    knn = KNeighborsClassifier(n_neighbors=k)

    #replacing the manual validations by cross validation scores
    #score -> returns an array contains the acc for each k value using cv
    scores = cross_val_score(
        estimator=knn,
        X= X_train,
        y= y_train,
        cv=kf ,
        scoring='accuracy'
    )
    print(scores)

    avg_accuracy = np.mean(scores)
    cv_accuracies.append(avg_accuracy)
```

then we got the best k value

```
#how to get the best k ? using the max avg_acc through the cv_accuracies
best_k_cv = k_values[np.argmax(cv_accuracies)]
print(f'the best k value:{best_k_cv} and its acc = { max(cv_accuracies):.3f}')
```

then we got the test acc based on the k _ value we got from the cv because it is being reliable most of the time as it is not biased to a specific split

also we normalized x_train to make the values on the same scale

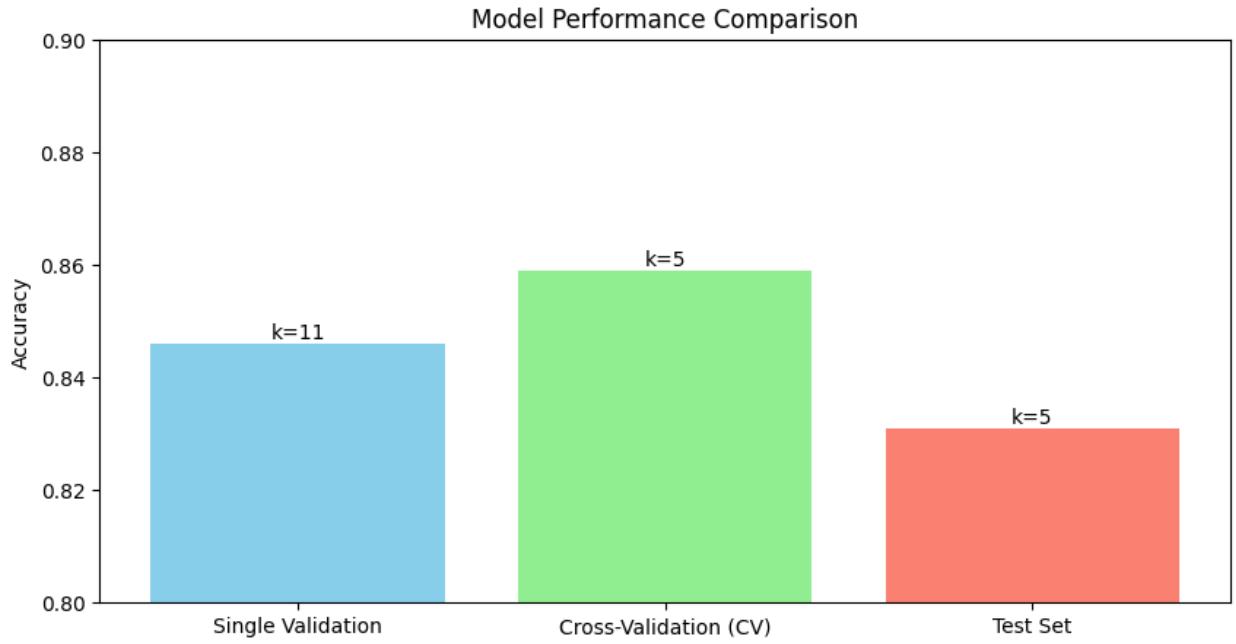
```
#max validation acc = 0.853 , k = 11
#max cv acc = 0.855  , k = 7 more reliable
#test set results -> weather the cv better or the single val set is better means
final evaluation
#train on all untest data
x_train_val = pd.concat([X_train , X_val] , axis=0 )
y_train_val = pd.concat([y_train , y_val] , axis = 0)

from sklearn.preprocessing import StandardScaler

scaler = StandardScaler()
x_train_val_s = scaler.fit_transform(x_train_val)
X_test_s = scaler.transform(X_test)
```

```python
#final model
final_knn = KNeighborsClassifier(n_neighbors= best_k_cv)
final_knn.fit(x_train_val_s , y_train_val)
test_acc  = final_knn.score(X_test_s , y_test)
print(f'the final test acc :{test_acc:.3f}')
```

we've got the final model to get the test set also and the acc was = 0.831

## the comparison

we made in the code also using visualization



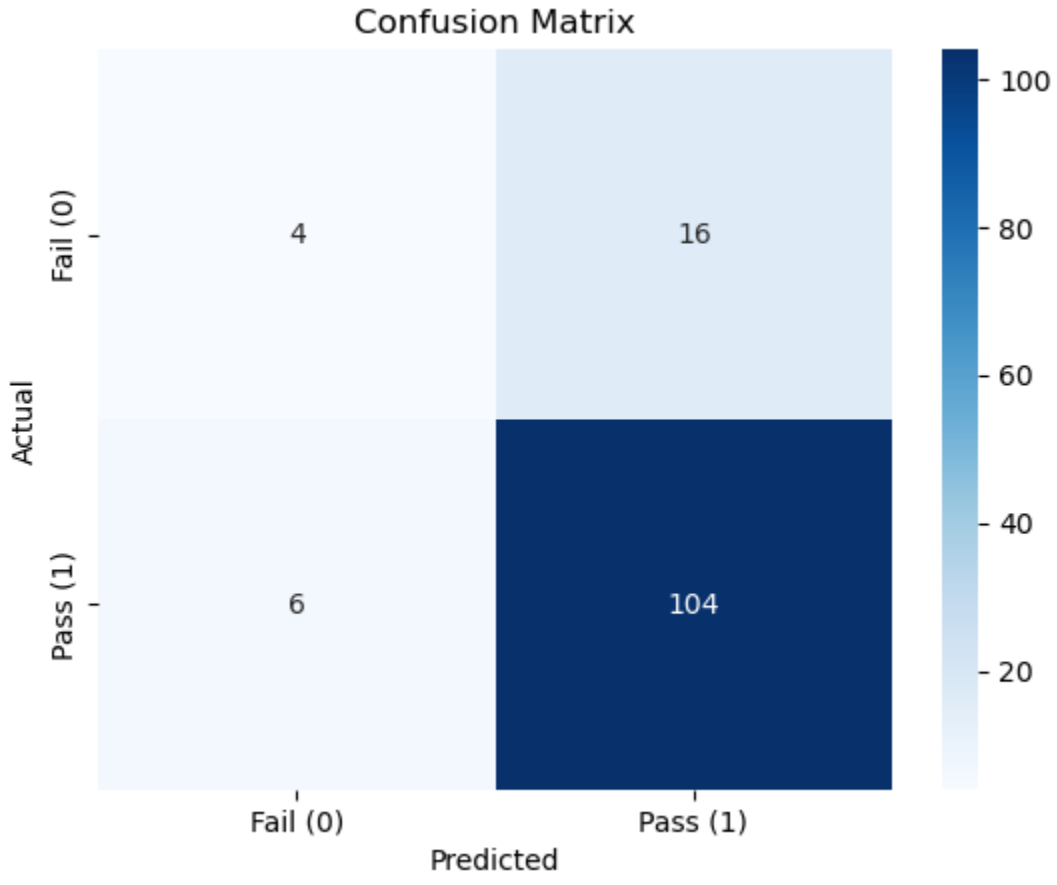

## Discussion :

now the cv is more realistic as it performs on all the splits and the results show that the both validation set and cv are estimate to the test result slightly optimistic , but the validation is less realistic beacause it comes from one split may this split by luck gives me this acc and another split gives less so i think cv was really a good choice because it indicates the performance whenever the split was and it has a low variance with the final test acc where var = |0.838 - 0.856| =0.018
means low variance

## Confusion Matrix :

we got the y_pred to get the acc score

```
# now the final selected model is knn with k = 5 and test acc = 0.838
from sklearn.metrics import classification_report , confusion_matrix
#final model = final_knn and start to get the y_pred

y_pred = final_knn.predict(X_test_s)
```



## Key insights:

about the class **Fail(0)** it has a too low recal means that the model misses about 80% of the students who where failed
and the percesion means that the model predict fail 40% correct of the time = 60 % wrong
all this problems may be because data class inbalance the  number of class pass is much more big than class fail this what is why the model predicts about class fails false -> we need to adjust cross validation method or the threshold of the pass col based on G3 we already tried so many times as explained in the code

about class **pass(1)** it has a too high recall means that the model misses only 5% of students who where passed
and the percesion is moderate , slightly high means that the model predict pass correct 87% of the time

## Model Improvement:

The model did not have an overfitting issue; however, the data set used was imbalanced in terms of passed to failed students. To adjust this issue, Feature Selection, using Random Forest to select the top 20 most important features. In addition, ADASYN was used to handle class imbalance.

Feature Selection Using Random Forest

```
# Use Random Forest to get feature importance
rf = RandomForestClassifier(random_state=42)
rf.fit(features_encoded, df['pass'])
feature_importance = pd.DataFrame({'Feature': features_encoded.columns, 'Importance': rf.feature_importances_})
feature_importance = feature_importance.sort_values(by='Importance', ascending=False)

# Select top 20 features (arbitrary threshold, can be tuned)
top_features = feature_importance['Feature'].head(20).values
features_selected = features_encoded[top_features]

print("Selected Features:\n", top_features)
```

Uses a Random Forest model to rank features by importance (rf.feature_importances_) and selects the top 20 features to reduce dimensionality.
**Selected Features**: Includes failures, absences, school_MS, higher_yes, famrel, etc.
**Why Feature Selection?**

- Reduces computational complexity for KNN, which is sensitive to high-dimensional data.
- Improves model interpretability and potentially reduces noise from less relevant features.
- Random Forest is robust for feature importance because it evaluates how much each feature contributes to reducing impurity across multiple trees.

## Handling Class Imbalance with ADASYN

```
# Apply ADASYN to training data to address class imbalance
adasyn = ADASYN(random_state=42)
X_train_adasyn, y_train_adasyn = adasyn.fit_resample(X_train, y_train)

print("Training set after ADASYN:", X_train_adasyn.shape, y_train_adasyn.shape)
print("Class distribution after ADASYN:\n", pd.Series(y_train_adasyn).value_counts())
```

**What is ADASYN?**

- **Adaptive Synthetic Sampling (ADASYN)** is an oversampling technique that generates synthetic samples for the minority class (fail=0) to balance the dataset.

**Why ADASYN?**

- The dataset is imbalanced: pass=1 (majority) significantly outnumbers fail=0 (minority). The test set distribution shows 110 pass vs. 20 fail.
- Imbalanced data can bias the model toward the majority class, leading to poor performance on the minority class (low recall for fail).
- ADASYN balances the training set by generating synthetic fail samples, increasing the minority class representation.

**Impact of ADASYN**:

- **Before ADASYN**: The training set had 389 samples with a skewed distribution (likely ~85% pass, ~15% fail, based on the overall dataset).
- **After ADASYN**: The training set grows to 667 samples, with a near-balanced distribution (338 fail, 329 pass).
- **Effect on Model**:
    - Improves the model's ability to learn patterns for the fail class, potentially increasing recall for fail.
    - However, synthetic samples may introduce noise or overfitting if not carefully tuned.
- **No Impact on Validation/Test**: ADASYN is applied only to the training set to avoid data leakage and ensure realistic evaluation.