Worksheet 12

Name: Zihan Li, Jingbo Wang, Yuchen Chao

UID: U83682995, U04536921, U51424608

Topics

- Introduction to Classification
- K Nearest Neighbors

Introduction to Classification

a) For the following examples, say whether they are or aren't an example of classification.

1. Predicting whether a student will be offered a job after graduating given their GPA.
2. Predicting how long it will take (in number of months) for a student to be offered a job after graduating, given their GPA.
3. Predicting the number of stars (1-5) a person will assign in their yelp review given the description they wrote in the review.
4. Predicting the number of births occuring in a specified minute.

1. This is an example of classification. The outcome is categorical: either the student will be offered a job or they won't be.
2. This is not an example of classification. Instead, it's a regression problem since the outcome is a continuous variable (number of months).
3. This is an example of classification. The outcome is categorical with 5 possible classes (1, 2, 3, 4, or 5 stars).
4. This is not an example of classification. It's a regression problem because the outcome (number of births in a minute) is continuous.

b) Given a dataset, how would you set things up such that you can both learn a model and get an idea of how this model might perform on data it has never seen?

Doing cross-validation by splitting the data into training, validation and testing sets. Tune the hyper-parameters and finally evaluate on the trained model.

c) In your own words, briefly explain:

- underfitting
- overfitting

and what signs to look out for for each.

1. underfitting Underfitting happens when the model is too simple and fail to learn well from the data. Signs to look out for: poor performance on both training and testing data.

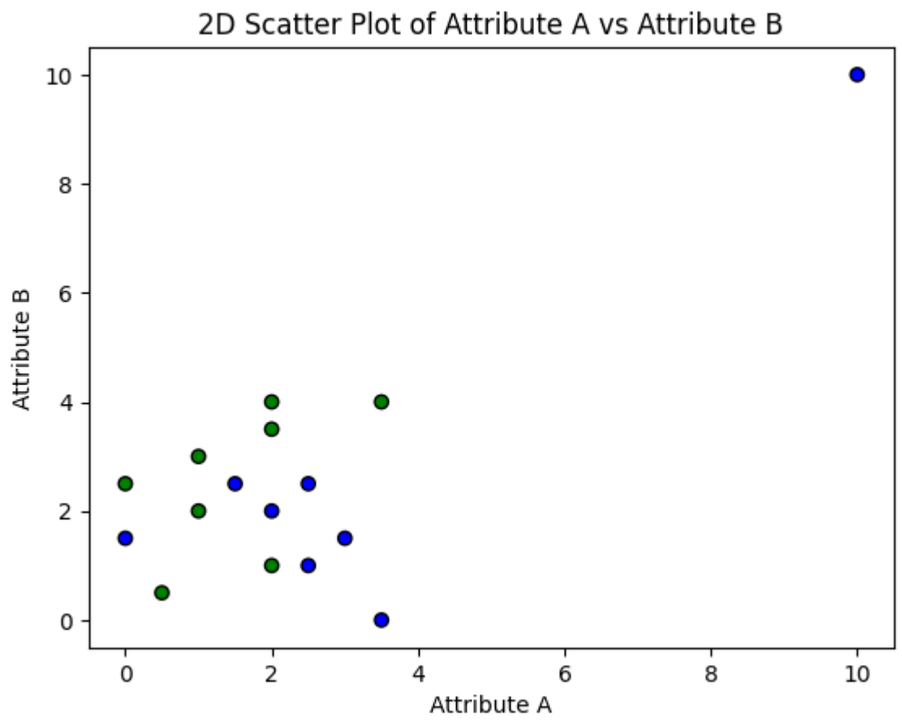
2. overfitting Overfitting happens when the model becomes too complex and too sensitive at the training data and became affected by noises from the training data. Signs to look out for: good enough performance on training data yet poor performance on testing data.

K Nearest Neighbors

In [ ]:
```python
import numpy as np
import matplotlib.pyplot as plt

data = {
    "Attribute A" : [3.5, 0, 1, 2.5, 2, 1.5, 2, 3.5, 1, 3, 2, 2, 2.5, 0.5, 0., 10],
    "Attribute B" : [4, 1.5, 2, 1, 3.5, 2.5, 1, 0, 3, 1.5, 4, 2, 2.5, 0.5, 2.5, 10],
    "Class" : [1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 0, 1, 1, 0],
}
```

a) Plot the data in a 2D plot coloring each scatter point one of two colors depending on its corresponding class.

In [ ]:
```python
colors = np.array([x for x in 'bgrcmyk'])
plt.scatter(data["Attribute A"], data["Attribute B"], color=colors[data["Class"]].tolist(), edgecolors='k')
plt.xlabel('Attribute A')
plt.ylabel('Attribute B')
plt.title('2D Scatter Plot of Attribute A vs Attribute B')
plt.show()
```



Outliers are points that lie far from the rest of the data. They are not necessarily invalid points however. Imagine sampling from a Normal Distribution with mean 10 and variance 1. You would expect most points you sample to be in the range [7, 13] but it's entirely possible to see 20 which, on average, should be very far from the rest of the points in the sample (unless we're VERY (un)lucky). These outliers can inhibit our ability to learn general patterns in the data since they are not representative of likely outcomes. They can still be useful in of themselves and can be analyzed in great depth depending on the problem at hand.

b) Are there any points in the dataset that could be outliers? If so, please remove them from the dataset.

Yes

In [ ]:
```python
# Convert data to numpy arrays for easier computation
attribute_a = np.array(data["Attribute A"])
attribute_b = np.array(data["Attribute B"])
classes = np.array(data["Class"])

# Calculate IQR for Attribute A
q1_a = np.percentile(attribute_a, 25)
q3_a = np.percentile(attribute_a, 75)
iqr_a = q3_a - q1_a
lower_bound_a = q1_a - 1.5 * iqr_a
upper_bound_a = q3_a + 1.5 * iqr_a

# Calculate IQR for Attribute B
```

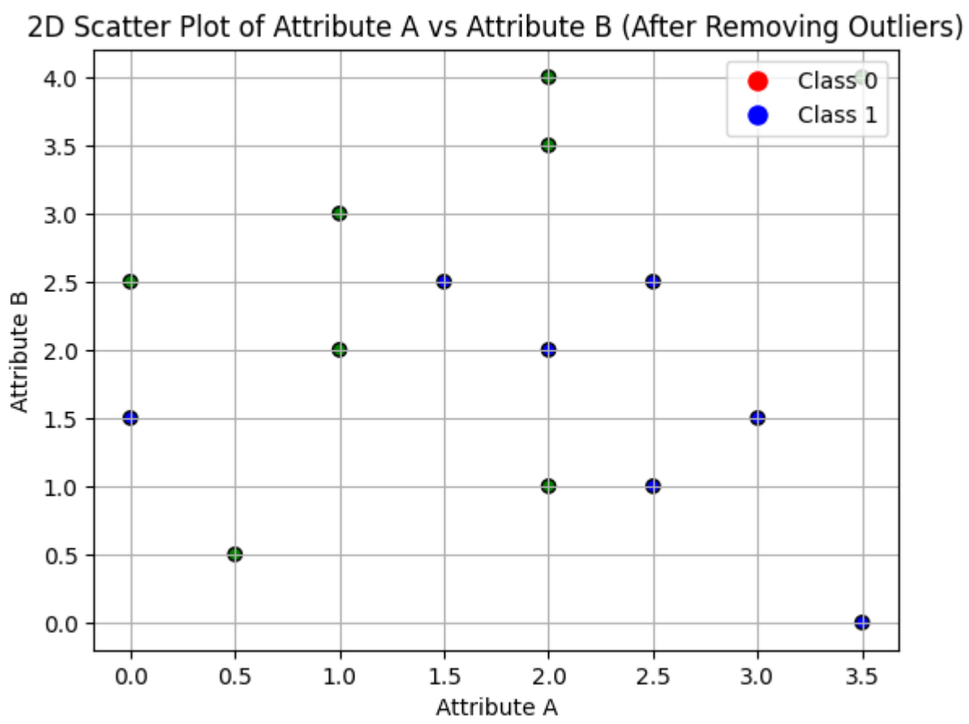
```
q1_b = np.percentile(attribute_b, 25)
q3_b = np.percentile(attribute_b, 75)
iqr_b = q3_b - q1_b
lower_bound_b = q1_b - 1.5 * iqr_b
upper_bound_b = q3_b + 1.5 * iqr_b

# Identify outliers
outliers_a = np.where((attribute_a < lower_bound_a) | (attribute_a > upper_bound_a))
outliers_b = np.where((attribute_b < lower_bound_b) | (attribute_b > upper_bound_b))

# Combine outliers
combined_outliers = np.union1d(outliers_a, outliers_b)

# Remove outliers from the dataset
attribute_a = np.delete(attribute_a, combined_outliers)
attribute_b = np.delete(attribute_b, combined_outliers)
classes = np.delete(classes, combined_outliers)

# Update data dictionary
data = {
    "Attribute A": attribute_a.tolist(),
    "Attribute B": attribute_b.tolist(),
    "Class": classes.tolist()
}
```

Noise points are points that could be considered invalid under the general trend in the data. These could be the result of actual errors in the data or randomness that we could attribute to oversimplification (for example if missing some information / feature about each point). Considering noise points in our model can often lead to overfitting.

c) Are there any points in the dataset that could be noise points?

No

In [ ]:
```
plt.scatter(data["Attribute A"], data["Attribute B"], color=colors[data["Class"]].tolist(), edgecolors='k')
plt.xlabel('Attribute A')
plt.ylabel('Attribute B')
plt.title('2D Scatter Plot of Attribute A vs Attribute B (After Removing Outliers)')
plt.legend(handles=[plt.Line2D([0], [0], marker='o', color='w', markerfacecolor='r', markersize=10, label='Class 0'),
                    plt.Line2D([0], [0], marker='o', color='w', markerfacecolor='b', markersize=10, label='Class 1')],
           loc='upper right')
plt.grid(True)
plt.show()
```

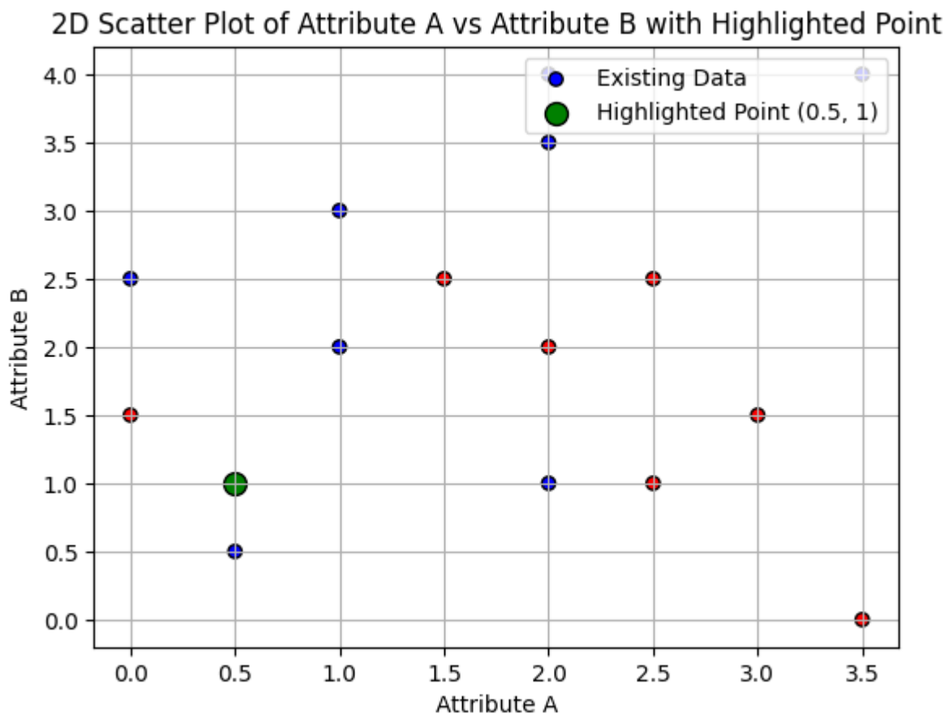


For the following point

A	B
0.5	1

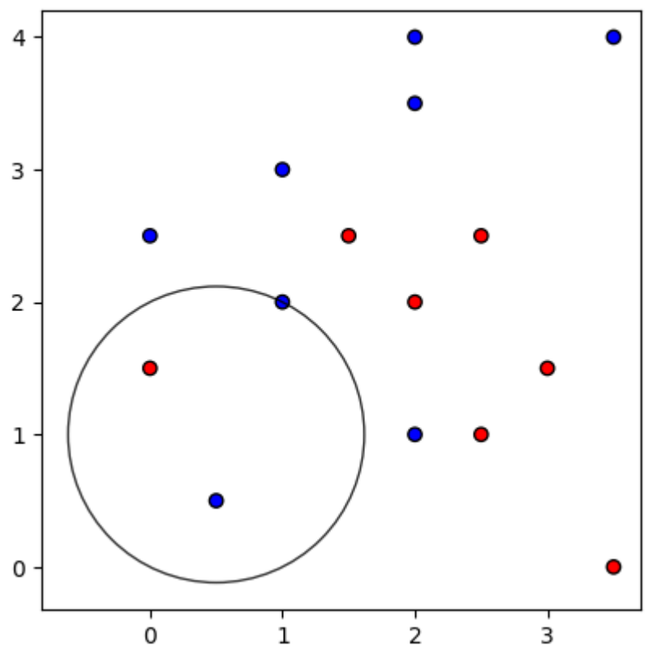
d) Plot it in a different color along with the rest of the points in the dataset.

In [ ]:
```
# Extract existing data
attribute_a = data["Attribute A"]
attribute_b = data["Attribute B"]
classes = data["Class"]

# Colors for the existing classes
colors = np.array(['r', 'b'])

# Plot the existing data points
plt.scatter(attribute_a, attribute_b, color=colors[classes].tolist(), edgecolors='k', label='Existing Data')

# Highlight the given point
plt.scatter(0.5, 1, color='g', edgecolors='k', s=100, label='Highlighted Point (0.5, 1)')

# Setting labels, title, and legend
plt.xlabel('Attribute A')
plt.ylabel('Attribute B')
plt.title('2D Scatter Plot of Attribute A vs Attribute B with Highlighted Point')
plt.legend(loc='upper right')
plt.grid(True)
plt.show()
```



e) Write a function to compute the Euclidean distance from it to all points in the dataset and pick the 3 closest points to it. In a scatter plot, draw a circle centered around the point with radius the distance of the farthest of the three points.

In [ ]:
```
def n_closest_to(example, n):
    distances = []
    for a, b in zip(attribute_a, attribute_b):
        dist = np.sqrt((a - example[0])**2 + (b - example[1])**2)
        distances.append(dist)

    # Sort distances and return the first n
    return sorted(distances)[:n]


location = (0.5, 1)
three_closest_distances = n_closest_to(location, 3)
radius = max(three_closest_distances)
_, axes = plt.subplots()
axes.scatter(attribute_a, attribute_b, color=colors[classes].tolist(), edgecolors='k')
cir = plt.Circle(location, radius, fill=False, alpha=0.8)
axes.add_patch(cir)
axes.set_aspect('equal') # necessary so that the circle is not oval
plt.show()
```



f) Write a function that takes the three points returned by your function in e) and returns the class that the majority of points have (break ties with a deterministic default class of your choosing). Print the class assigned to this new point by your function.

In [ ]:
```
def majority(points):
    # Extract classes of the given points
    classes_of_points = [data["Class"][attribute_a.index(point[0])] for point in points]

    # Count occurrences of each class
    count_0 = classes_of_points.count(0)
    count_1 = classes_of_points.count(1)

    # Return class with majority or default in case of tie
    if count_0 > count_1:
        return 0
    elif count_1 > count_0:
        return 1
    else:
        return 0  # default class in case of a tie

# Identify the three closest points to the example
example_point = (0.5, 1)
distances = [np.sqrt((a - example_point[0])**2 + (b - example_point[1])**2) for a, b in zip(attribute_a, attribute_b)]
three_closest_indices = np.argsort(distances)[:3]
three_closest_points = [(attribute_a[i], attribute_b[i]) for i in three_closest_indices]

# Get and print the majority class
assigned_class = majority(three_closest_points)
print(f"The class assigned to the point {example_point} is: {assigned_class}")
```

```
The class assigned to the point (0.5, 1) is: 1
```

g) Re-using the functions from e) and f), you should be able to assign a class to any new point. In this exercise we will implement Leave-one-out cross validiation in order to evaluate the performance of our model.

For each point in the dataset:

- consider that point as your test set and the rest of the data as your training set
- classify that point using the training set
- keep track of whether you were correct with the use of a counter

Once you've iterated through the entire dataset, divide the counter by the number of points in the dataset to report an overall testing accuracy.

In [ ]:
```
def classify_point(point, training_attribute_a, training_attribute_b):
    # Compute distances to all points in the training set
    distances = [np.sqrt((a - point[0])**2 + (b - point[1])**2) for a, b in zip(training_attribute_a, training_attribute_b)]
    # Get indices of the three closest points in the training set
    three_closest_indices = np.argsort(distances)[:3]
```

```
    # Extract three closest points
    three_closest_points = [(training_attribute_a[i], training_attribute_b[i]) for i in three_closest_indices]
    # Return the predicted class using the majority function
    return majority(three_closest_points)

count = 0
for i in range(len(data["Class"])):
    actual_class = data["Class"][i]
    # Exclude the current test point to create the training set
    training_attribute_a = [a for j, a in enumerate(data["Attribute A"]) if j != i]
    training_attribute_b = [b for j, b in enumerate(data["Attribute B"]) if j != i]

    prediction = classify_point((data["Attribute A"][i], data["Attribute B"][i]), training_attribute_a, training_attribute_b)

    if prediction == actual_class:
        count += 1

overall_accuracy = count / len(data["Class"])
print(f"Overall accuracy = {overall_accuracy:.2f}")
```

```
Overall accuracy = 0.60
```

Challenge Problem

For this question we will re-use the "mnist_784" dataset.

a) Begin by creating a training and testing datasest from our dataset, with a 80-20 ratio, and random_state=1. You can use the `train_test_split` function from sklearn. By holding out a portion of the dataset we can evaluate how our model generalizes to unseen data (i.e. data it did not learn from).

In [ ]:
```
from sklearn.datasets import fetch_openml
from sklearn.model_selection import train_test_split

# Load the mnist_784 dataset with parser set to 'auto'
X, y = fetch_openml('mnist_784', version=1, return_X_y=True, parser='auto')

# Split the dataset into training and testing sets with an 80-20 ratio
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=1)

# Output the shapes of the resulting datasets to verify the split
print("Training set features shape:", X_train.shape)
print("Training set labels shape:", y_train.shape)
print("Testing set features shape:", X_test.shape)
print("Testing set labels shape:", y_test.shape)
```

```
Training set features shape: (56000, 784)
Training set labels shape: (56000,)
Testing set features shape: (14000, 784)
Testing set labels shape: (14000,)
```

b) For K ranging from 1 to 20:

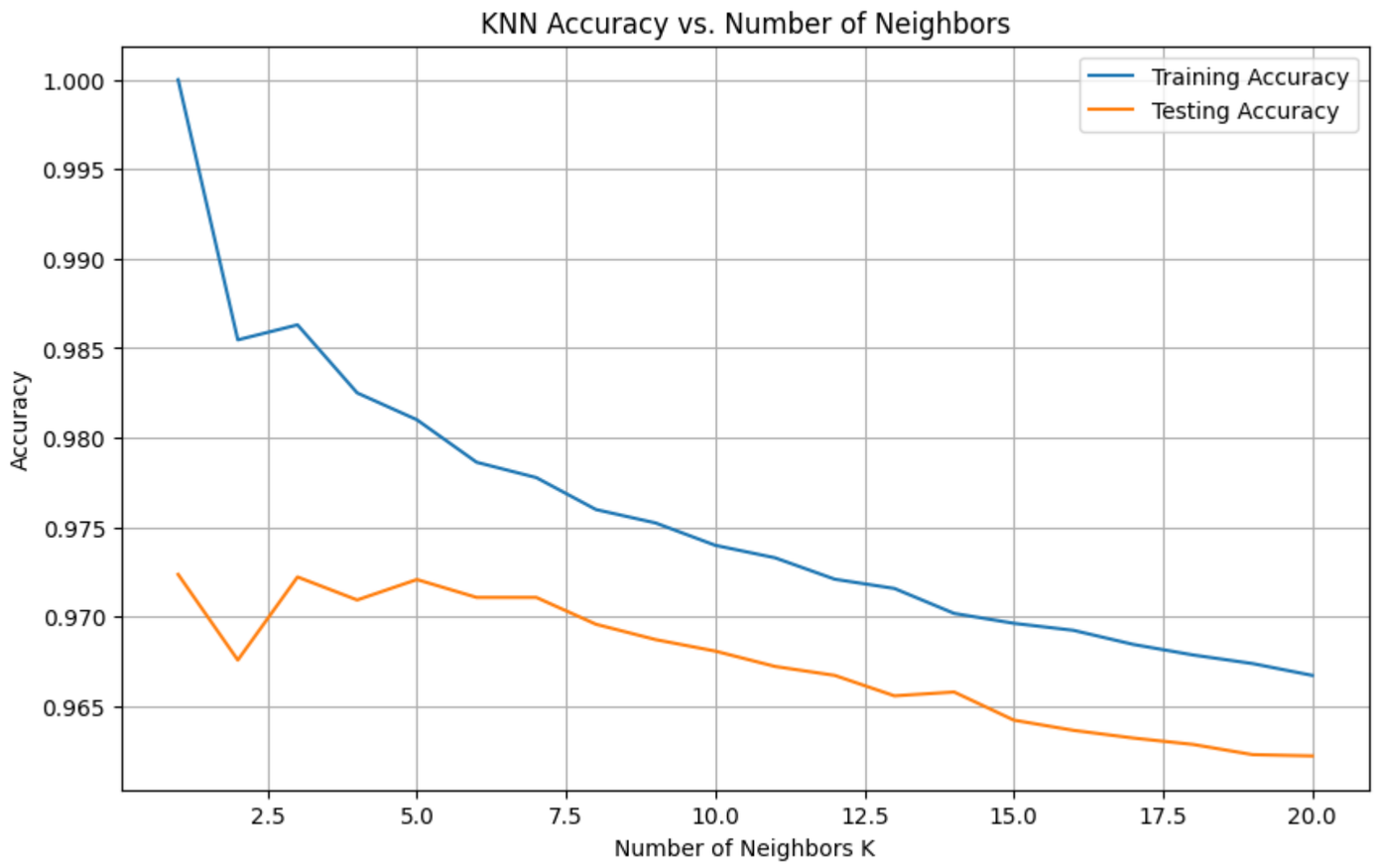
1. train a KNN on the training data
2. record the training and testing accuracy

Plot a graph of the training and testing set accuracy as a function of the number of neighbors K (on the same plot). Which value of K is optimal? Briefly explain.

In [ ]:
```
from tqdm import tqdm
```

In [ ]:
```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import accuracy_score

# Initialize lists to store accuracies
training_accuracies = []
testing_accuracies = []

# Train a KNN classifier for K from 1 to 20
for K in tqdm(range(1, 21), desc="Training KNN"):
    knn = KNeighborsClassifier(n_neighbors=K, n_jobs=-1) # Use all available cores
    knn.fit(X_train, y_train)

    # Record the training accuracy
    y_pred_train = knn.predict(X_train)
    training_accuracy = accuracy_score(y_train, y_pred_train)
    training_accuracies.append(training_accuracy)

    # Record the testing accuracy
    y_pred_test = knn.predict(X_test)
    testing_accuracy = accuracy_score(y_test, y_pred_test)
    testing_accuracies.append(testing_accuracy)

# Plotting the accuracies
plt.figure(figsize=(10, 6))
plt.plot(range(1, 21), training_accuracies, label='Training Accuracy')
plt.plot(range(1, 21), testing_accuracies, label='Testing Accuracy')
plt.xlabel('Number of Neighbors K')
plt.ylabel('Accuracy')
plt.title('KNN Accuracy vs. Number of Neighbors')
plt.legend()
plt.grid(True)
plt.show()
```

```
Training KNN:   0%|          | 0/20 [00:00<?, ?it/s]Training KNN: 100%|██████████| 20/20 [32:45<00:00, 98.30s/it]
```

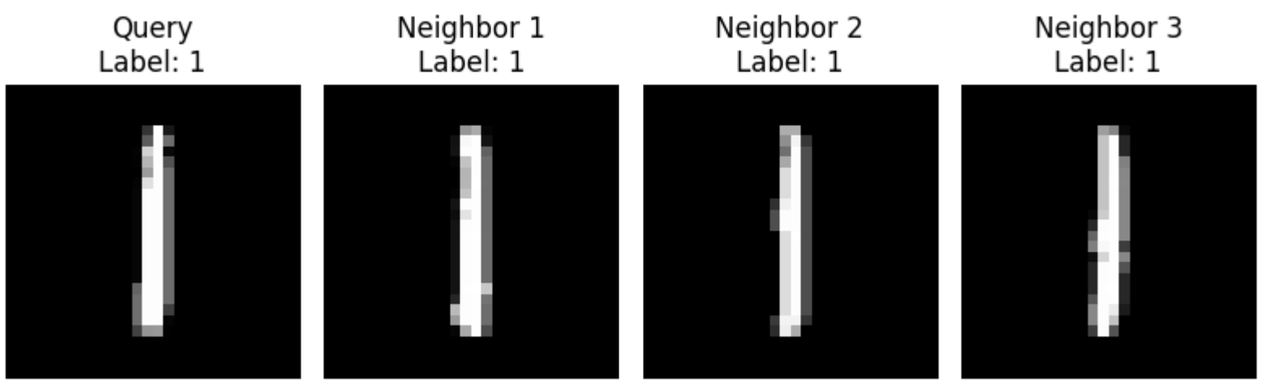


The optimal value of K is usually the value that maximizes test accuracy while maintaining a reasonable gap between training and test accuracy to avoid overfitting. Overfitting manifests itself as high training accuracy but much lower test accuracy, which indicates that the model has learned the training data too closely and does not generalize well to new data.

c) Using the best model from b), pick an image at random and plot it next to its K nearest neighbors

In [ ]:
```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.neighbors import KNeighborsClassifier
import random

# Assuming X_train, X_test, y_train, y_test are pandas DataFrame or Series
# Let's say the best K we found from part b) is K_best
K_best = 3  # Replace 3 with your actual best K value found from part b)

# Train the KNN model on the training set with the best K
knn = KNeighborsClassifier(n_neighbors=K_best)
knn.fit(X_train, y_train)

# Randomly pick an image from the test set
random_index = random.randint(0, X_test.shape[0] - 1)
query_image = X_test.iloc[random_index].values  # Use .iloc for pandas
query_label = y_test.iloc[random_index]  # Use .iloc for pandas

# Use the KNN model to find the K nearest neighbors of the query image
distances, indices = knn.kneighbors([query_image])

# Plot the query image
plt.figure(figsize=(2 * (K_best + 1), 4))
plt.subplot(1, K_best + 1, 1)
plt.imshow(query_image.reshape(28, 28), cmap='gray')
plt.title(f"Query\nLabel: {query_label}")
plt.axis('off')

# Plot each of the K nearest neighbors
for i, index in enumerate(indices[0], start=2):
    neighbor_image = X_train.iloc[index].values  # Use .iloc for pandas
    neighbor_label = y_train.iloc[index]  # Use .iloc for pandas
    plt.subplot(1, K_best + 1, i)
    plt.imshow(neighbor_image.reshape(28, 28), cmap='gray')
    plt.title(f"Neighbor {i-1}\nLabel: {neighbor_label}")
    plt.axis('off')

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

```
/Library/Frameworks/Python.framework/Versions/3.8/lib/python3.8/site-packages/sklearn/base.py:439: UserWarning: X does not have valid feature names, but KNeighborsClassifier
was fitted with feature names
  warnings.warn(
```

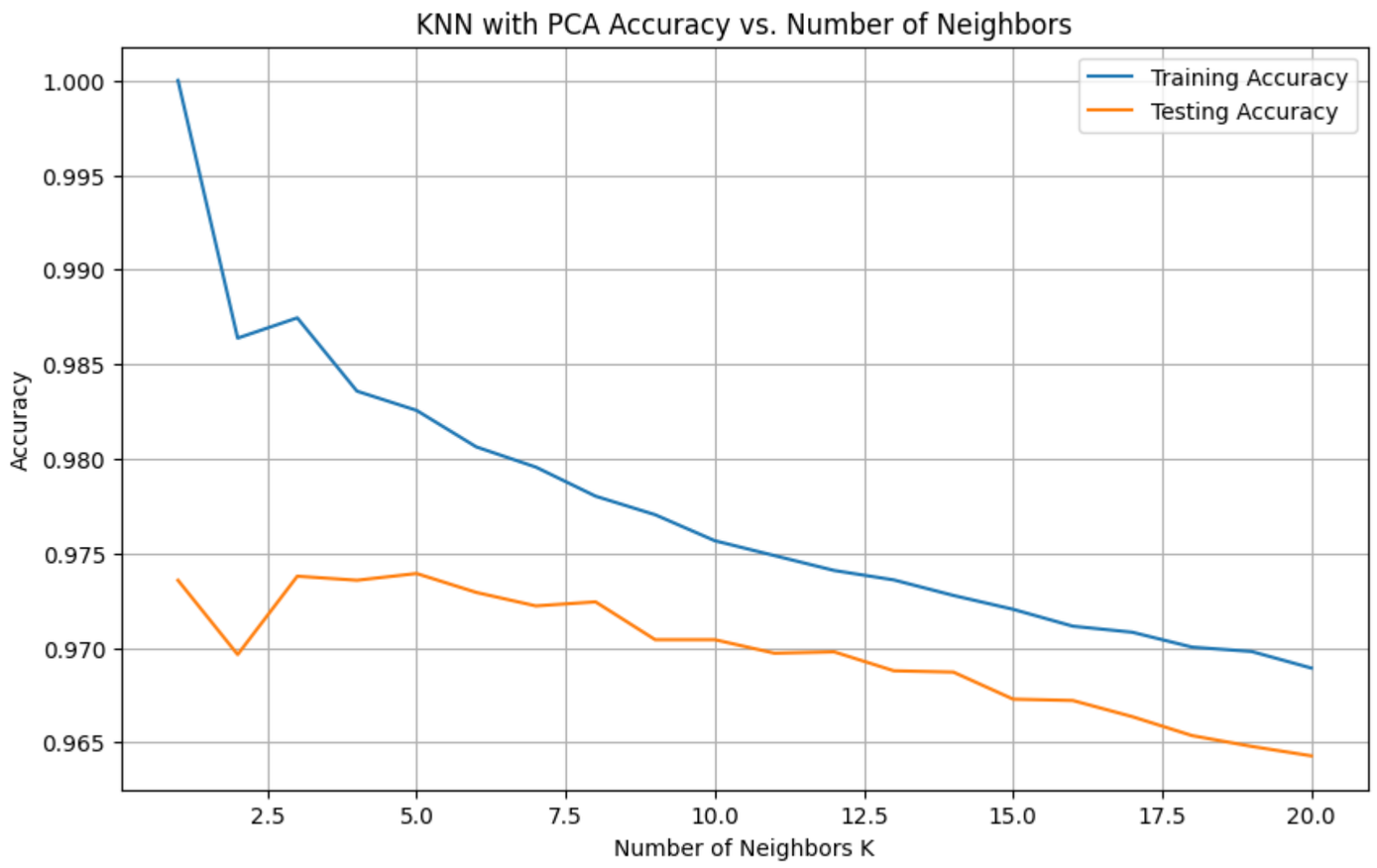


d) Using a dimensionality reduction technique discussed in class, reduce the dimensionality of the dataset before applying a KNN model. Repeat b) and discuss similarities and differences to the previous model. Briefly discuss your choice of dimension and why you think the performance / accuracy of the model has changed.

In [ ]:
```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.decomposition import PCA
from sklearn.neighbors import KNeighborsClassifier
from sklearn.pipeline import make_pipeline
from sklearn.metrics import accuracy_score
from tqdm import tqdm

# Assuming X_train, X_test, y_train, y_test are already defined

# Initialize lists to store accuracies
training_accuracies = []
testing_accuracies = []

# Number of PCA components (e.g., .95 for 95% variance)
n_components = 0.95

# Train a KNN classifier for K from 1 to 20, integrated with PCA
for K in tqdm(range(1, 21), desc="Training KNN with PCA"):
    # Create a pipeline with PCA and KNN
    model = make_pipeline(PCA(n_components=n_components), KNeighborsClassifier(n_neighbors=K,  n_jobs=-1))

    # Fit the model
    model.fit(X_train, y_train)
```

```
    # Record the training accuracy
    y_pred_train = model.predict(X_train)
    training_accuracy = accuracy_score(y_train, y_pred_train)
    training_accuracies.append(training_accuracy)

    # Record the testing accuracy
    y_pred_test = model.predict(X_test)
    testing_accuracy = accuracy_score(y_test, y_pred_test)
    testing_accuracies.append(testing_accuracy)

# Plotting the accuracies
plt.figure(figsize=(10, 6))
plt.plot(range(1, 21), training_accuracies, label='Training Accuracy')
plt.plot(range(1, 21), testing_accuracies, label='Testing Accuracy')
plt.xlabel('Number of Neighbors K')
plt.ylabel('Accuracy')
plt.title('KNN with PCA Accuracy vs. Number of Neighbors')
plt.legend()
plt.grid(True)
plt.show()
```

```
Training KNN with PCA: 100%|██████████| 20/20 [04:52<00:00, 14.62s/it]
```



After implementing the above operations, the performance and accuracy of the KNN model with PCA is compared with the original KNN model without dimensionality reduction. Consider the following:

1. **Performance/accuracy variation**: models with PCA may exhibit different performance and accuracy due to dimensionality reduction. In some cases, accuracy may decrease slightly due to loss of information, but in other cases, accuracy may actually increase if PCA helps to remove noise from the data.
2. **Computation time**: The computation time of a KNN model with PCA is usually shorter due to the reduced number of dimensions.
3. **Choice of dimensions**: The choice of the number of PCA components (dimensions) is critical. It is a balance between retaining enough information (variance) and achieving computational efficiency. The variation in performance is directly related to the extent to which the selected dimensions capture the essential information of the original data.

Midterm Prep (Part 1)

Compete in the Titanic Data Science Competition on Kaggle: https://www.kaggle.com/c/titanic

Requirements:

1. Add at least 2 new features to the dataset (explain your reasoning below)
2. Use KNN (and only KNN) to predict survival
3. Explain your process below and choice of K
4. Make a submission to the competition and provide a link to your submission below.
5. Show your code below

In [ ]:
```
import pandas as pd
from sklearn.model_selection import train_test_split, cross_val_score
from sklearn.neighbors import KNeighborsClassifier
from sklearn.preprocessing import StandardScaler
from sklearn.impute import SimpleImputer
from sklearn.pipeline import make_pipeline
from sklearn.compose import ColumnTransformer
from sklearn.preprocessing import OneHotEncoder
from sklearn.model_selection import GridSearchCV

# Load the dataset
train_data = pd.read_csv('train.csv')
test_data = pd.read_csv('test.csv')

# Feature engineering
train_data['FamilySize'] = train_data['SibSp'] + train_data['Parch'] + 1
test_data['FamilySize'] = test_data['SibSp'] + test_data['Parch'] + 1

train_data['Title'] = train_data['Name'].str.extract(' ([A-Za-z]+)\.', expand=False)
test_data['Title'] = test_data['Name'].str.extract(' ([A-Za-z]+)\.', expand=False)

# Preprocessing
numerical_cols = ['Age', 'Fare', 'FamilySize']
numerical_transformer = make_pipeline(SimpleImputer(strategy='median'), StandardScaler())

categorical_cols = ['Sex', 'Embarked', 'Title']
categorical_transformer = OneHotEncoder(handle_unknown='ignore')

preprocessor = ColumnTransformer(
    transformers=[
        ('num', numerical_transformer, numerical_cols),
        ('cat', categorical_transformer, categorical_cols)
    ])

# Splitting the training data for training and validation
X = train_data.drop(['Survived', 'PassengerId'], axis=1)
y = train_data['Survived']
X_train, X_valid, y_train, y_valid = train_test_split(X, y, test_size=0.2, random_state=0)

# Define the model and pipeline
knn = KNeighborsClassifier()
pipeline = make_pipeline(preprocessor, knn)

# Grid search for the best K
param_grid = {'kneighborsclassifier__n_neighbors': range(1, 50)}
grid_search = GridSearchCV(pipeline, param_grid, cv=5)
```

```
grid_search.fit(X_train, y_train)

print(f"Best K: {grid_search.best_params_['kneighborsclassifier__n_neighbors']}")
print(f"Best cross-validation score: {grid_search.best_score_}")

# Evaluate the model using the best K
best_knn = grid_search.best_estimator_
predictions = best_knn.predict(X_valid)
print(f"Validation accuracy: {accuracy_score(y_valid, predictions)}")

# Prepare the test data and make predictions for submission
# Note: The test data should be preprocessed in the same way as the training data
test_preds = best_knn.predict(test_data.drop(['PassengerId'], axis=1))

# Generate submission file
output = pd.DataFrame({'PassengerId': test_data['PassengerId'], 'Survived': test_preds})
output.to_csv('my_submission.csv', index=False)
```

```
Best K: 19
Best cross-validation score: 0.8202797202797203
Validation accuracy: 0.8100558659217877
```

1. Data preparation

Feature Engineering:

- **FamilySize**: Created by adding the number of siblings/spouses (SibSp) and the number of parents/children (Parch) for each passenger, plus one passenger's own number. The rationale is that a passenger's family size may affect his or her chances of survival.
- **Title**: Taken from the name of each passenger. Titles may reflect social status, gender, and marital status, which may affect survival chances.

Preprocessing:

Numerical features: Age, fare and family size are standardized and missing values are estimated using the median. Normalization is crucial for KNN as it relies on distance calculation. **Categorical features**: Sex, Embarked and Title were coded once and converted into a format suitable for modeling. This step is necessary since KNN cannot handle categorized data directly.

2 Model training and validation

Splitting the data:

- The dataset is split into training and validation sets to evaluate the performance of the model and avoid overfitting.

Create pipeline:

- Build pipelines to simplify preprocessing and model training. This ensures that all preprocessing steps are applied consistently in the training and validation phases.

3. Optimize K

Grid search:

- To determine the optimal value of K, a grid search was performed over a range of possible K values (from 1 to 50). This method is systematically trained and evaluated by cross-validation of models with different K values to ensure that the choice of K values is robust and not overly dependent on the specificity of the training data.
- Cross-validation:
- Cross-validation is used during the grid search to evaluate the performance of the model at each K value. This method splits the training data into several folds, trains the model on some folds, and then validates the model on other folds. This process is repeated several times and the average performance across all folds of data is used to evaluate each K value.

Selection of K:

- This method balances the bias-variance tradeoff inherent in K-value selection: smaller K-values may cause the model to capture noise (overfitting), while larger K-values may make the data too smooth to capture important patterns (underfitting).

https://www.kaggle.com/competitions/titanic/leaderboard?search=JingboWangBU