worksheet_12

March 10, 2024

1 Worksheet 12

Name: Youxuan Ma

UID: U23330522

1.0.1 Topics

- Introduction to Classification
- K Nearest Neighbors

1.0.2 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.

- Examples (1) and (3) demonstrate classification tasks, because they involve categorizing inputs into discrete labels based on the given features.
- In contrast, examples (2) and (4) are not classification tasks as they involve predicting continuous outcomes.

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?

I would set things up step by step as followings:

1. Dataset Preliminary Steps

- Clean the Data
- Feature Selection
- Feature Scaling/Normalization

2. Splitting the Dataset

- Training and Test Split:
  - Divide dataset into a training set and a test set. A common split ratio is 70% (training) to 30% (test), although this can vary based on the size and specifics of the dataset.

3. Cross-validation

- K-Fold Cross-validation:
  - One of the most common methods is K-Fold cross-validation. Here, the data is randomly divided into 'K' equal folds. The model is trained on 'K-1' folds and tested on the remaining fold. This process is repeated 'K' times (folds), with each fold being used exactly once as the test set. Finally, the model's performance is averaged over the 'K' trials to get a more comprehensive idea of how the model might perform on unseen data.

4. Model Training and Validation

- Hyperparameter Tuning
- Evaluation on the Test Set
- Using the Evaluation to Predict Performance

5. Conclusion and Iteration

- Based on the model's performance on the test set, revisit some of the earlier steps to improve the model further and iteratively.

c) In your own words, briefly explain:

- underfitting
- overfitting

and what signs to look out for for each.

1. Underfitting occurs when a model is too simple to capture the underlying structure of the data. It can't even perform well on the training data, which means it's also unable to generalize well to unseen data.

- Signs to look for:
  1. Poor Performance on Training Data
  2. Simplistic Model Relative to the Data Complexity
  3. Comparable Poor Performance on Both Training and Validation/Test Data
  4. No Improvement or Marginal Improvement with More Training

2. Overfitting occurs when a model learns both the underlying trends and the noise in the training dataset to an extent where it performs exceptionally well on the training data but fails to generalize to unseen data. Essentially, the model becomes too complex, capturing spurious correlations that don't actually exist in the broader dataset.

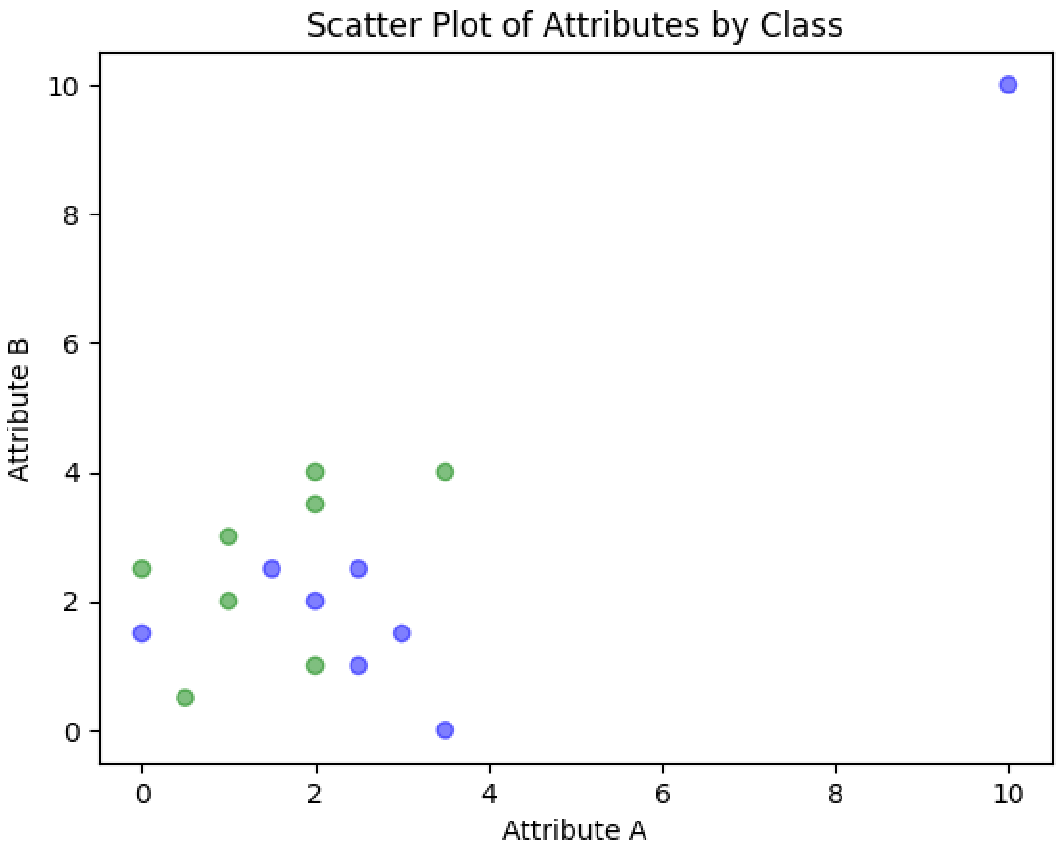
- Signs to look for:
  1. High Training Accuracy but Low Validation/Test Accuracy
  2. Complex Model Structure with Many Parameters
  3. Significant Improvement in Training Performance with No Equivalent Improvement on Validation/Test Data
  4. Sensitivity to Small Fluctuations in the Training Data
  5. Model Learns Detailed Features Irrelevant to the Task

1.0.3 K Nearest Neighbors

[48]:
```
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.

[49]:
```
colors = np.array([x for x in 'bgrcmyk'])
plt.scatter(np.array(data["Attribute A"]), np.array(data["Attribute B"]),␣
  ↪color=colors[data["Class"]].tolist(), alpha=0.5)
plt.title('Scatter Plot of Attributes by Class')
plt.xlabel('Attribute A')
plt.ylabel('Attribute B')
plt.show()
```



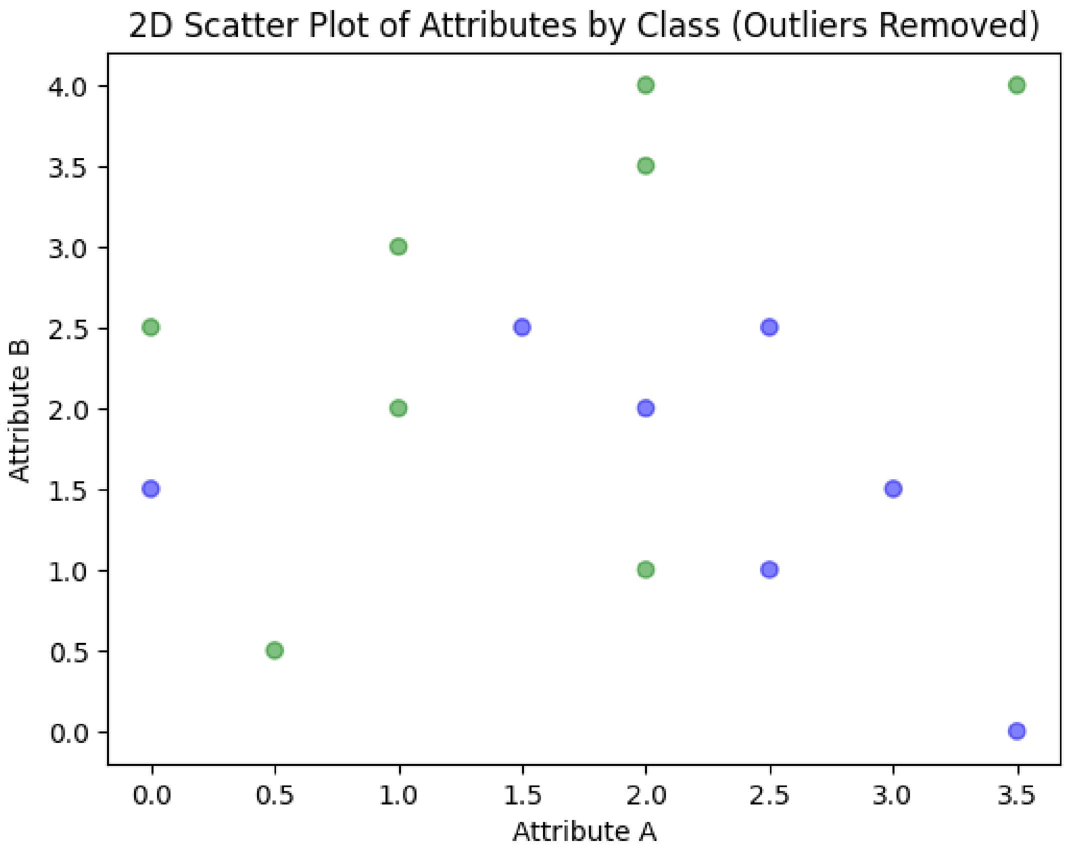
Outliers are points that lie far from the rest of the data. They are not necessarily invalid points however. Imagine sampling from a Normal Distribution with mean 10 and variance 1. You would expect most points you sample to be in the range [7, 13] but it's entirely possible to see 20 which, on average, 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.

As we see, the point (10, 10) is very likely an outlier, and we will remove it from the dataset as follow.

[50]:
```
attribute_a = np.array(data["Attribute A"])
attribute_b = np.array(data["Attribute B"])
classes = np.array(data["Class"])

Q1_a, Q3_a = np.percentile(attribute_a, [25, 75])
IQR_a = Q3_a - Q1_a

Q1_b, Q3_b = np.percentile(attribute_b, [25, 75])
IQR_b = Q3_b - Q1_b

outlier_indices_a = np.where((attribute_a < (Q1_a - 1.5 * IQR_a)) |␣
↪(attribute_a > (Q3_a + 1.5 * IQR_a)))
outlier_indices_b = np.where((attribute_b < (Q1_b - 1.5 * IQR_b)) |␣
↪(attribute_b > (Q3_b + 1.5 * IQR_b)))

outlier_indices = np.unique(np.concatenate((outlier_indices_a,␣
↪outlier_indices_b)))

print("Outlier Points:")
for index in outlier_indices:
    print(f"Index: {index}, Attribute A: {data['Attribute A'][index]},␣
↪Attribute B: {data['Attribute B'][index]}, Class: {data['Class'][index]}")

clean_attribute_a = np.delete(attribute_a, outlier_indices)
clean_attribute_b = np.delete(attribute_b, outlier_indices)
clean_classes = np.delete(classes, outlier_indices)

plt.scatter(clean_attribute_a, clean_attribute_b, color=colors[clean_classes],␣
↪alpha=0.5)
plt.title('2D Scatter Plot of Attributes by Class (Outliers Removed)')
plt.xlabel('Attribute A')
plt.ylabel('Attribute B')
plt.show()
```

```
Outlier Points:
Index: 15, Attribute A: 10, Attribute B: 10, Class: 0
```



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?

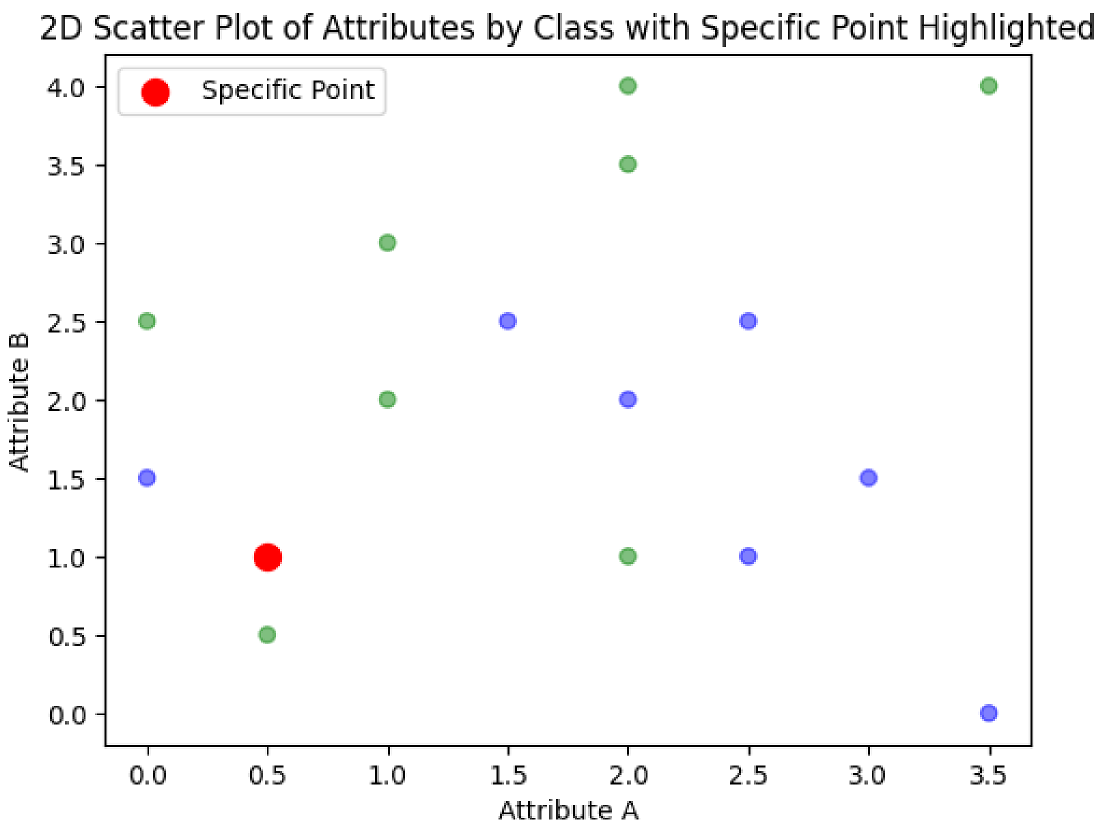
The point (0, 1.5) might be a noise point, since it's far from the points of its own class and is very close to the points of the other class.

For the following point

A	B
0.5	1

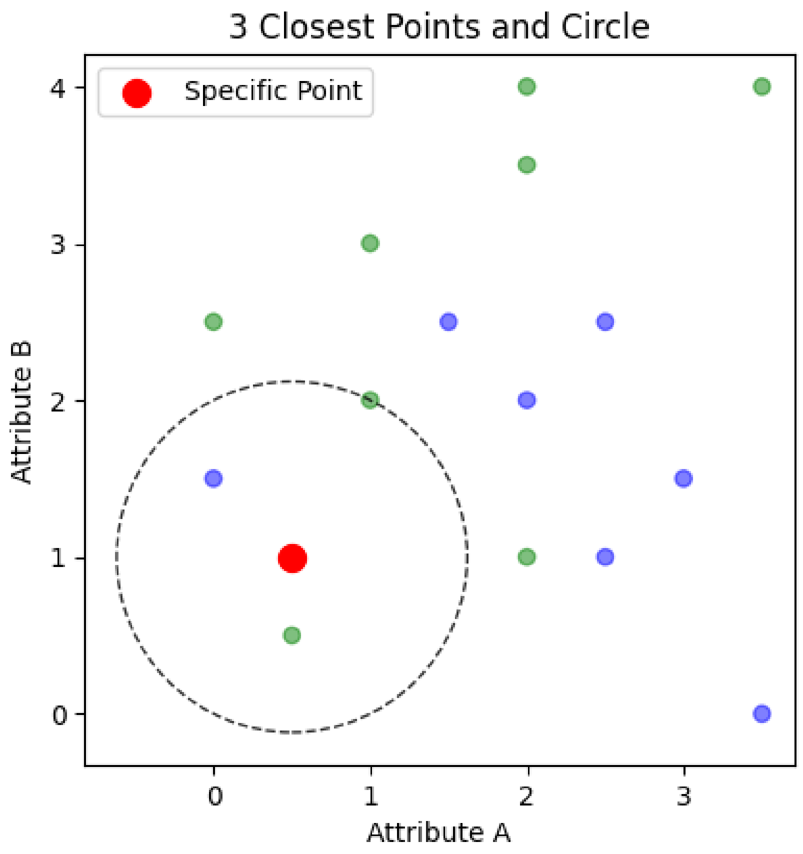
d) Plot it in a different color along with the rest of the points in the dataset.

[51]:
```
specific_point_a = 0.5
specific_point_b = 1

plt.scatter(clean_attribute_a, clean_attribute_b, color=colors[clean_classes],␣
  ↪alpha=0.5)

plt.scatter(specific_point_a, specific_point_b, color='red', s=100,␣
  ↪label='Specific Point')

plt.title('2D Scatter Plot of Attributes by Class with Specific Point␣
  ↪Highlighted')
plt.xlabel('Attribute A')
plt.ylabel('Attribute B')
plt.legend()
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.

```
[52]: dataset_points = np.column_stack((clean_attribute_a, clean_attribute_b))

      def n_closest_to(example, n, dataset):
          distances = np.sqrt(((dataset - example) ** 2).sum(axis=1))
          closest_indices = np.argsort(distances)[:n]
          return closest_indices, distances[closest_indices]

      location = (0.5, 1)
      n = 3

      closest_indices, closest_distances = n_closest_to(location, n, dataset_points)

      radius = np.max(closest_distances)

      _, axes = plt.subplots()

      axes.scatter(clean_attribute_a, clean_attribute_b, color=colors[clean_classes],␣
       ↪alpha=0.5)

      axes.scatter(*location, color='r', s=100, label='Specific Point')

      cir = plt.Circle(location, radius, color='black', fill=False, alpha=0.8,␣
       ↪linestyle='--')
      axes.add_patch(cir)
      axes.set_aspect('equal')
      plt.legend()
      plt.title('3 Closest Points and Circle')
      plt.xlabel('Attribute A')
      plt.ylabel('Attribute B')

      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.

[53]:
```
def majority(indices):

    closest_classes = clean_classes[indices]
    class_counts = np.bincount(closest_classes)
    max_count = np.max(class_counts)
    tied_classes = np.where(class_counts == max_count)[0]

    if len(tied_classes) > 1:
        selected_class = tied_classes[0]
    else:
        selected_class = np.argmax(class_counts)

    return selected_class

maj_class = majority(closest_indices)
```

```python
print(f"The majority class among the closest points to {location} is:␣
  ↪{maj_class}.")
print(f"Therefore, the class assigned to this new point is: {maj_class}.")
```

```
The majority class among the closest points to (0.5, 1) is: 1.
Therefore, the class assigned to this new point 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.

[54]:
```python
count = 0
num_data = len(dataset_points)
k = 3

for i in range(num_data):
    training_points = np.delete(dataset_points, i, axis=0)
    training_classes = np.delete(clean_classes, i)

    actual_class = clean_classes[i]
    current_point = dataset_points[i]

    cls_ind, _ = n_closest_to(current_point, k, training_points)

    prediction = majority(cls_ind)

    if prediction == actual_class:
        count += 1

accuracy = count / num_data

print(f"overall accuracy = {accuracy:.2f}")
```

```
overall accuracy = 0.60
```

1.1 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).

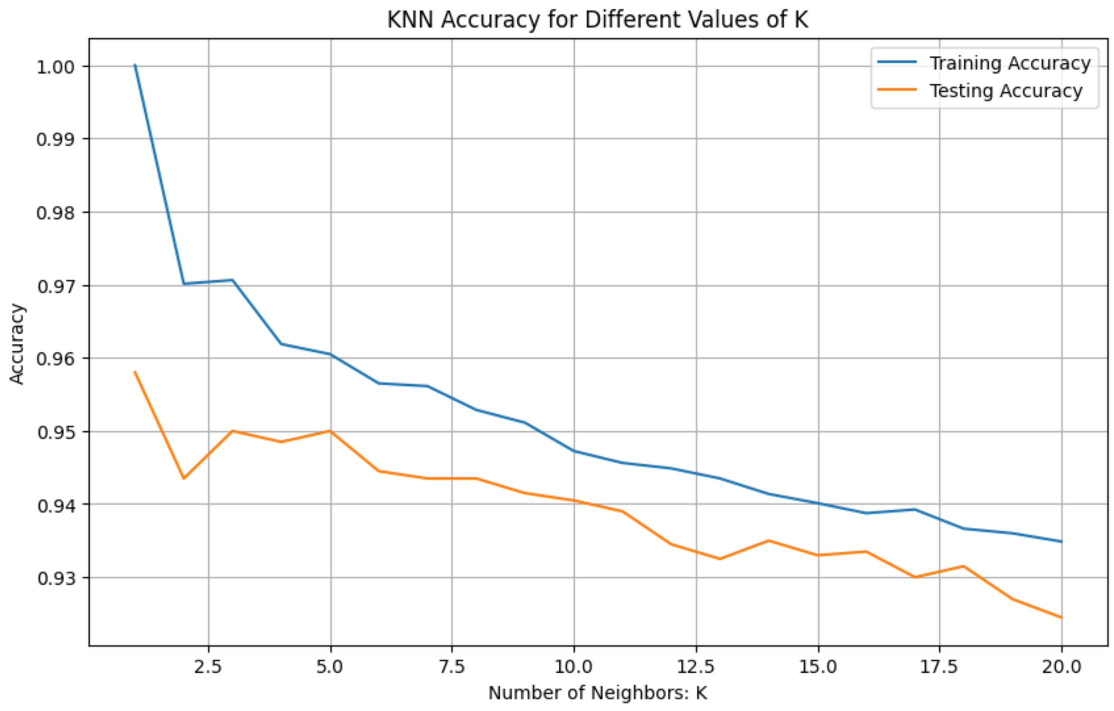
[55]:
```python
from sklearn.model_selection import train_test_split
from sklearn.datasets import fetch_openml
from sklearn.utils import resample

mnist = fetch_openml('mnist_784', version=1)

X, y = np.array(mnist["data"]), np.array(mnist["target"])

# Take a random sample of 10,000 data points,
# since the entire dataset is too large for my computer to handle
sample_X, sample_y = resample(X, y, n_samples=10000, random_state=42)

X_train, X_test, y_train, y_test = train_test_split(sample_X, sample_y,␣
 ↪test_size=0.20, random_state=1)
```

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.

[56]:
```python
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import accuracy_score

training_accuracies = []
testing_accuracies = []

k_range = range(1, 21)

for k in k_range:

    knn = KNeighborsClassifier(n_neighbors=k)
    knn.fit(X_train, y_train)

    y_train_pred = knn.predict(X_train)
    training_accuracy = accuracy_score(y_train, y_train_pred)
    training_accuracies.append(training_accuracy)

    y_test_pred = knn.predict(X_test)
    testing_accuracy = accuracy_score(y_test, y_test_pred)
    testing_accuracies.append(testing_accuracy)

plt.figure(figsize=(10, 6))
plt.plot(k_range, training_accuracies, label='Training Accuracy')
plt.plot(k_range, testing_accuracies, label='Testing Accuracy')
plt.xlabel('Number of Neighbors: K')
plt.ylabel('Accuracy')
plt.title('KNN Accuracy for Different Values of K')
plt.legend()
plt.grid(True)
plt.show()

# Find the K value with the highest testing accuracy
optimal_k = testing_accuracies.index(max(testing_accuracies)) + 1
max_accuracy = max(testing_accuracies)

print(f"The optimal K value is {optimal_k}, with a testing accuracy of␣
 ↪{max_accuracy:.4f}.")
```



```
The optimal K value is 1, with a testing accuracy of 0.9580.
```

As we can see from the result, the optimal K value is 1, as it yields both the highest testing accuracy and training accuracy.

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

[57]:
```
knn_model = KNeighborsClassifier(n_neighbors=optimal_k)
knn_model.fit(X_train, y_train)
```

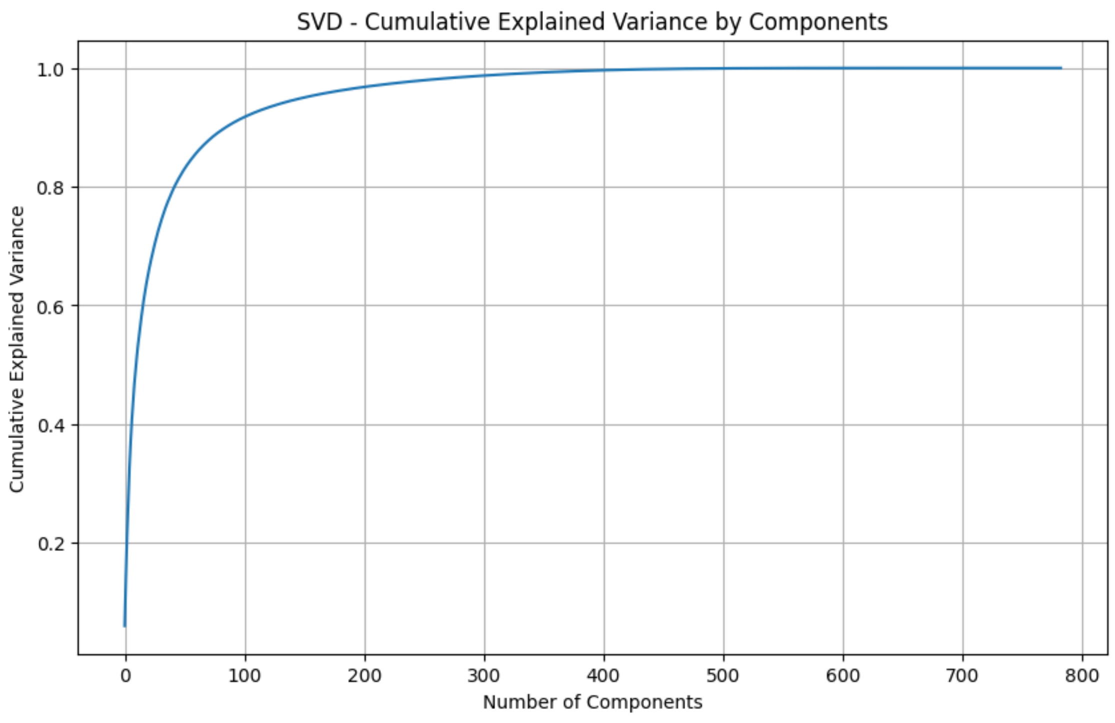
```
random_index = np.random.randint(0, sample_X.shape[0])
random_image = sample_X[random_index]

random_image_reshaped = random_image.reshape(1, -1)

distances, indices = knn_model.kneighbors(random_image_reshaped)

fig, axes = plt.subplots(nrows=1, ncols=optimal_k+1, figsize=(15, 3))

axes[0].imshow(random_image.reshape(28, 28), cmap='gray')
axes[0].set_title('Selected Image')
axes[0].axis('off')

for i, neighbor_idx in enumerate(indices[0]):
    neighbor_image = X_train[neighbor_idx].reshape(28, 28)
    axes[i+1].imshow(neighbor_image, cmap='gray')
    axes[i+1].set_title(f'Neighbor {i+1}')
    axes[i+1].axis('off')

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

Selected Image
Neighbor 1

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.

[58]:
```
from sklearn.decomposition import TruncatedSVD

# Apply SVD with a high number of components to analyze the singular values
svd = TruncatedSVD(n_components=min(X_train.shape)-1)
X_train_svd = svd.fit_transform(X_train)
```

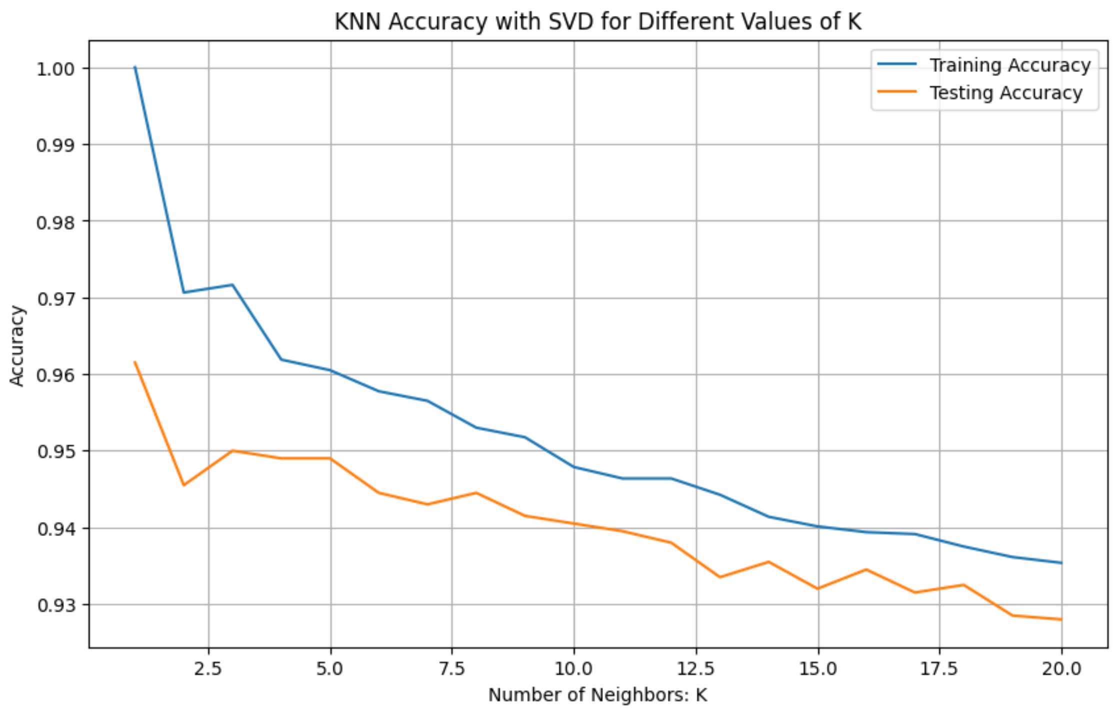
```
cumulative_variance = np.cumsum(svd.explained_variance_ratio_)
differences = np.diff(cumulative_variance)

# Plot the explained variance as a function of the number of singular values
plt.figure(figsize=(10, 6))
plt.plot(np.cumsum(svd.explained_variance_ratio_))
plt.xlabel('Number of Components')
plt.ylabel('Cumulative Explained Variance')
plt.title('SVD - Cumulative Explained Variance by Components')
plt.grid(True)
plt.show()

elbow_point = np.argwhere(differences < np.mean(differences) * 0.1)[0][0]

print(f"The elbow point is at {elbow_point} components, with a cumulative␣
 ↪explained variance of {cumulative_variance[elbow_point]:.4f}.")
```



```
The elbow point is at 310 components, with a cumulative explained variance of
0.9885.
```

[59]:
```
from sklearn.pipeline import make_pipeline

dimensionality_reduction = TruncatedSVD(n_components=elbow_point)
```

```python
X_train_reduced = dimensionality_reduction.fit_transform(X_train)
X_test_reduced = dimensionality_reduction.transform(X_test)

training_accuracies = []
testing_accuracies = []
k_range = range(1, 21)

for k in k_range:
    knn = KNeighborsClassifier(n_neighbors=k)
    knn.fit(X_train_reduced, y_train)

    y_train_pred = knn.predict(X_train_reduced)
    training_accuracy = accuracy_score(y_train, y_train_pred)
    training_accuracies.append(training_accuracy)

    y_test_pred = knn.predict(X_test_reduced)
    testing_accuracy = accuracy_score(y_test, y_test_pred)
    testing_accuracies.append(testing_accuracy)


plt.figure(figsize=(10, 6))
plt.plot(k_range, training_accuracies, label='Training Accuracy')
plt.plot(k_range, testing_accuracies, label='Testing Accuracy')
plt.xlabel('Number of Neighbors: K')
plt.ylabel('Accuracy')
plt.title('KNN Accuracy with SVD for Different Values of K')
plt.legend()
plt.grid(True)
plt.show()

optimal_k = testing_accuracies.index(max(testing_accuracies)) + 1
print(f"The optimal K value is: {optimal_k}")

knn_model = KNeighborsClassifier(n_neighbors=optimal_k)
model = make_pipeline(dimensionality_reduction, knn_model)
model.fit(X_train, y_train)
```



```
The optimal K value is: 1
```

```
[59]: Pipeline(steps=[('truncatedsvd', TruncatedSVD(n_components=310)),
                      ('kneighborsclassifier', KNeighborsClassifier(n_neighbors=1))])
```

1. The similarity is that the optimal value of K doesn't change.
2. The differences are that the testing accuracy improved and the training and prediction times shortened.
3. My choice of dimensions (n_components) in SVD was based on a balance between preserving enough information (variance) and improving computational efficiency. To that end, I used the elbow point in the explained variance plot, where the addition of new components no longer contributes significantly to capturing more variance, as the optimal dimensions.
4. As we see from the result, the performance and accuracy improved. I think the reason for that is very likely due to noise reduction and focusing the model on the most informative features.

1.2 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

[60]:
```python
import pandas as pd

# Load the datasets
train_df = pd.read_csv('./train.csv')
test_df = pd.read_csv('./test.csv')

# Extract Title from Name
train_df['Title'] = train_df['Name'].apply(lambda x: x.split(',')[1].split('.
↪')[0].strip())
test_df['Title'] = test_df['Name'].apply(lambda x: x.split(',')[1].split('.
↪')[0].strip())

# Calculate Family Size
train_df['FamilySize'] = train_df['SibSp'] + train_df['Parch'] + 1
test_df['FamilySize'] = test_df['SibSp'] + test_df['Parch'] + 1

# Determine if the passenger is alone
train_df['IsAlone'] = train_df['FamilySize'].apply(lambda x: 1 if x == 1 else 0)
test_df['IsAlone'] = test_df['FamilySize'].apply(lambda x: 1 if x == 1 else 0)

# Display the first few rows of the training dataset to verify the new features
train_df.head()
```

[60]:
```
   PassengerId  Survived  Pclass  \
0            1         0       3
1            2         1       1
2            3         1       3
3            4         1       1
4            5         0       3

                                                Name     Sex   Age  SibSp  \
0                            Braund, Mr. Owen Harris    male  22.0      1
1  Cumings, Mrs. John Bradley (Florence Briggs Th…  female  38.0      1
2                             Heikkinen, Miss. Laina  female  26.0      0
3       Futrelle, Mrs. Jacques Heath (Lily May Peel)  female  35.0      1
4                           Allen, Mr. William Henry    male  35.0      0

   Parch            Ticket     Fare Cabin Embarked Title  FamilySize  IsAlone
0      0         A/5 21171   7.2500   NaN        S    Mr           2        0
1      0          PC 17599  71.2833   C85        C   Mrs           2        0
2      0  STON/O2. 3101282   7.9250   NaN        S  Miss           1        1
3      0            113803  53.1000  C123        S   Mrs           2        0
4      0            373450   8.0500   NaN        S    Mr           1        1
```

1.2.1 Reasonings for Added Features

1. **Title:** Titles reflect social status and can indicate priority for lifeboats and rescue efforts. For example, women (Miss, Mrs) and higher social ranks (like officers) might have been given priority.
2. **FamilySize:** The size of a passenger's family aboard could impact their survival. Small families might have been more agile, whereas individuals from larger families might have prioritized staying together over individual survival.
3. **IsAlone:** This binary indicator helps us isolate the effect of social bonds on survival. Traveling alone might have made it easier to secure a spot on a lifeboat, as single passengers had fewer obligations to others, potentially allowing for quicker decision-making.

[61]:
```
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler, LabelEncoder
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import accuracy_score

import numpy as np

# Filling missing values

# Age imputation based on Pclass and Sex
age_impute_group = train_df.groupby(['Pclass', 'Sex'])['Age'].median()

def impute_age(row):
    if pd.isnull(row['Age']):
        return age_impute_group.loc[row['Pclass'], row['Sex']]
    else:
        return row['Age']

train_df['Age'] = train_df.apply(impute_age, axis=1)

train_df['Embarked'] = train_df['Embarked'].fillna(train_df['Embarked'].
  ↪mode()[0])

# Fare imputation based on Pclass and Embarked
fare_impute_group = train_df.groupby(['Pclass', 'Embarked'])['Fare'].median()

def impute_fare(row):
    if pd.isnull(row['Fare']):
        return fare_impute_group.loc[row['Pclass'], row['Embarked']]
    else:
        return row['Fare']

train_df['Fare'] = train_df.apply(impute_fare, axis=1)

# Encoding categorical variables
encoder = LabelEncoder()
```

```
train_df['Sex'] = encoder.fit_transform(train_df['Sex'])
train_df['Embarked'] = encoder.fit_transform(train_df['Embarked'])
train_df['Title'] = encoder.fit_transform(train_df['Title'])

# Preparing features and target variable
X = train_df.drop(['Survived', 'PassengerId', 'Name', 'Ticket', 'Cabin'],␣
 ↪axis=1)
y = train_df['Survived']

# Splitting the dataset into training and validation sets
X_train, X_val, y_train, y_val = train_test_split(X, y, test_size=0.2,␣
 ↪random_state=42)

# Standardizing the features
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_val_scaled = scaler.transform(X_val)

# Range of K to try
k_values = range(1, 51)
scores = []

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

    knn.fit(X_train_scaled, y_train)

    y_pred = knn.predict(X_val_scaled)

    accuracy = accuracy_score(y_val, y_pred)
    scores.append(accuracy)

# Finding the optimal K
optimal_k = k_values[scores.index(max(scores))]
optimal_accuracy = max(scores)
print(f"\nThe optimal number of neighbors is: {optimal_k}, with an accuracy of:␣
 ↪{optimal_accuracy:.4f}.")
```

```
The optimal number of neighbors is: 5, with an accuracy of: 0.8156.
```

1.3 Process Explaination and the Choice of K

1.3.1 Data Preparation and Feature Engineering

Imputation Strategy

- **Age**: I proposed imputing missing ages based on median values within groups defined by `Pclass` and `Sex`, considering these factors likely influenced age distributions aboard the Titanic.

- **Embarked**: Given the minimal number of missing values, I filled these with the mode, representing the most common embarkation port.
- **Fare**: Missing fares were imputed based on the median fare within the same `Pclass` and `Embarked` group, assuming ticket prices correlated with these categories.

Feature Engineering I introduced new features (`Title`, `FamilySize`, and `IsAlone`) to capture social status, family dynamics, and solitude, hypothesizing their potential impact on survival chances.

1.3.2 Determining Optimal K

I determined the optimal `K` by training models with different `K` values and evaluating their accuracy on a validation set. This empirical approach allowed me to balance bias and variance, aiming for the highest predictive performance. The specific `K` value chosen, K = 5, reflects a peak in validation accuracy.

1.3.3 Training and Predictions

After preprocessing the data and identifying the optimal `K`, I trained the KNN model on the entire training dataset, scaled to normalize feature magnitudes for distance calculation. The model was then used to predict survival on the similarly preprocessed and scaled test dataset.

[62]:
```
knn_optimal = KNeighborsClassifier(n_neighbors=optimal_k)
knn_optimal.fit(X_train_scaled, y_train)

# Prepare the test dataset
age_impute_group = test_df.groupby(['Pclass', 'Sex'])['Age'].median()

def impute_age(row):
    if pd.isnull(row['Age']):
        return age_impute_group.loc[row['Pclass'], row['Sex']]
    else:
        return row['Age']

test_df['Age'] = test_df.apply(impute_age, axis=1)
test_df['Embarked'] = test_df['Embarked'].fillna(test_df['Embarked'].mode()[0])

fare_impute_group = test_df.groupby(['Pclass', 'Embarked'])['Fare'].median()

def impute_fare(row):
    if pd.isnull(row['Fare']):
        return fare_impute_group.loc[row['Pclass'], row['Embarked']]
    else:
        return row['Fare']

test_df['Fare'] = test_df.apply(impute_fare, axis=1)
```

```
encoder = LabelEncoder()
test_df['Sex'] = encoder.fit_transform(test_df['Sex'])
test_df['Embarked'] = encoder.fit_transform(test_df['Embarked'])
test_df['Title'] = encoder.fit_transform(test_df['Title'])

# Preprocess the test dataset
X_test_dataset = test_df.drop(['PassengerId', 'Name', 'Ticket', 'Cabin'],␣
 ↪axis=1)

scaler = StandardScaler()
X_test_scaled = scaler.fit_transform(X_test_dataset)

# Make predictions on the test dataset
predictions = knn_optimal.predict(X_test_scaled)

# Create and save the submission file
submission_df = pd.DataFrame({'PassengerId': test_df['PassengerId'], 'Survived':
 ↪ predictions})

submission_file_path = './titanic_knn_prediction.csv'
submission_df.to_csv(submission_file_path, index=False)
```

1.3.4 My Competition Submission Link: https://www.kaggle.com/code/markma316/titanic-prediction