Assignment 2

Supervised Machine Learning Fundamentals

Name: Pranshul Bhatnagar

NetID: Pb251

Exercise 1 - Conceptual Questions on Supervised Learning I

[4 points]

For each part below, indicate whether we would generally expect the performance of a flexible statistical learning method to be *better* or *worse* than an inflexible method. Justify your answer.

1.1. The sample size n is extremely large, and the number of predictors p is small.

Ans. Better. When n is very large, there is enough data to reliably estimate complex patterns without severe overfitting. Since p is small, the model does not suffer much from the curse of dimensionality, allowing flexible methods to take advantage of the large sample size to capture subtle relationships.

1.2. The number of predictors p is extremely large, and the number of observations n is small.

Ans. Worse. With many predictors and few observations, flexible methods are prone to overfitting the noise in the data. Inflexible methods impose stronger structure or regularization, which helps control variance and typically leads to better generalization in this setting.

1.3. The relationship between the predictors and response is highly non-linear.

Ans. Better. Flexible methods are designed to capture complex, non-linear relationships, whereas inflexible methods may suffer from high bias if the true relationship is far from linear or simple.

1.4. The variance of the error terms, i.e. $\sigma^2 = Var(\epsilon)$, is extremely high.

Ans. Worse. When the noise level is very high, flexible methods are more likely to fit the noise rather than the underlying signal, increasing variance and hurting out-of-sample performance. Inflexible methods, with their higher bias but lower variance, tend to be more robust in high-noise settings.

Exercise 2 - Conceptual Questions on Supervised Learning II

[6 points]

For each of the following, (i) explain if each scenario is a classification or regression problem AND why, (ii) indicate whether we are most interested in inference or prediction for that problem AND why, and (iii) provide the sample size n and number of predictors p indicated for each scenario.

2.1. We collect a set of data on the top 500 firms in the US. For each firm we record profit, number of employees, industry and the CEO salary. We are interested in understanding which factors affect CEO salary.

Ans.
(i) This is a regression problem because the response variable (CEO salary) is quantitative/continuous
(ii) We are primarily interested in inference, since the goal is to understand which factors affect CEO salary and how they do so, rather than simply predicting salary values.
(iii) Sample size:500; Number of predictor:3

2.2. We are considering launching a new product and wish to know whether it will be a success or a failure. We collect data on 20 similar products that were previously launched. For each product we have recorded whether it was a success or failure, price charged for the product, marketing budget, competition price, and ten other variables.

Ans.
(i) Classification problem because the response variable is categorical with two outcomes: success or failure.
(ii) Prediction, since the goal is to predict whether a new product will succeed or fail.
(iii) Sample size: 20; Number of predictor: 13

2.3. We are interested in predicting the % change in the US dollar in relation to the weekly changes in the world stock markets. Hence we collect weekly data for all of 2012. For each week we record the % change in the dollar, the % change in the US market, the % change in the British market, and the % change in the German market.

Ans.
(i) Regression problem because the response variable (% change in the US dollar) is quantitative.
(ii) Prediction as the objective is to predict future dollar changes based on stock market movements.
(iii) Sample size: 52; Number of predictor: 3

Exercise 3 - Classification using KNN

[6 points]

The table below provides a training dataset containing six observations (a.k.a. samples) ($n = 6$) each with three predictors (a.k.a. features) ($p = 3$), and one qualitative response variable (a.k.a. target).

Table 1. Training dataset with $n = 6$ observations in $p = 3$ dimensions with a categorical response, y

Obs.	x_1	x_2	x_3	y
1	0	3	0	Red
2	2	0	0	Red
3	0	1	3	Red
4	0	1	2	Blue
5	-1	0	1	Blue
6	1	1	1	Red

We want to use the above training dataset to make a prediction, $\hat{y}$, for an unlabeled test data observation where $x_1 = x_2 = x_3 = 0$ using K-nearest neighbors. You are given some code below to get you started. *Note: coding is only required for part (a), for (b)-(d) please provide your reasoning based on your answer to part (a).*

3.1. Compute the Euclidean distance between each observation and the test point, $x_1 = x_2 = x_3 = 0$. Present your answer in a table similar in style to Table 1 with observations 1-6 as the row headers.

In [ ]:
```
import numpy as np
import pandas as pd

X = np.array([[0, 3, 0], [2, 0, 0], [0, 1, 3], [0, 1, 2], [-1, 0, 1], [1, 1, 1]])
y = np.array(["r", "r", "r", "b", "b", "r"])

# 3.1
# test point
x_test = np.array([0, 0, 0])

# calculate distance
distance = np.sqrt(np.sum((X - x_test) ** 2, axis=1))

# make table
df = pd.DataFrame({"Obs": np.arange(1, len(distance) + 1), "x₁": X[:, 0], "x₂": X
df = df.set_index("Obs")
print("Question 3.1:")
print(df)
```

```
Question 3.1:
     x₁  x₂  x₃  y
Obs
1     0   3   0  r
2     2   0   0  r
3     0   1   3  r
4     0   1   2  b
5    -1   0   1  b
6     1   1   1  r
```

3.2

When K=1, the test point is Blue. This is because this is the nearest point to test point(0, 0, 0) is blue(5).

3.3

When K=3, the test point is Red. This is because the nearst 3 points are blue(5), red(6), and red(2), and due to the majority vote, the answer should be red.

3.4

Small K. This is because when Bayes decision boundary is nonlinear, we need to use flexible method to solve the problem. And small k can make the model more flexible.

Exercise 4 - Build your own classification algorithm

[18 points]

> **Note**
>
> Data for this exercise can be downloaded here

4.1. Build a working version of a binary KNN classifier using the skeleton code below. We'll use the `sklearn` convention that a supervised learning algorithm has the methods `fit` which trains your algorithm (for KNN that means storing the data) and `predict` which identifies the K nearest neighbors and determines the most common class among those K neighbors. *Note: Most classification algorithms typically also have a method `predict_proba` which outputs the confidence score of each prediction, but we will explore that in a later assignment. Please use `NumPy` to implement euclidean distance function.*

In [2]:
```
# 4.1
# Skeleton code for part (a) to write your own kNN classifier


class Knn:
    # k-Nearest Neighbor class object for classification training and testing
    def __init__(self):
        self.x_train = None
        self.y_train = None

    def fit(self, x, y):
        # Save the training data to properties of this class
        self.x_train = x
        self.y_train = y

    def predict(self, x, k):
        y_hat = []  # Variable to store the estimated class label for
        # Calculate the distance from each vector in x to the training data
        x = np.array(x)
        for x_test in x:
            distances = np.sqrt(np.sum((self.x_train - x_test) ** 2, axis=1))
            knn_index = np.argsort(distances)[:k]
            knn_labels = self.y_train[knn_index]

            # majority vote
            values, counts = np.unique(knn_labels, return_counts=True)
            y_hat.append(values[np.argmax(counts)])
        y_hat = np.array(y_hat)
```

```
        # Return the estimated targets
        return y_hat


# Metric of overall classification accuracy
#  (a more general function, sklearn.metrics.accuracy_score, is also available)
def accuracy(y, y_hat):
    nvalues = len(y)
    accuracy = sum(y == y_hat) / nvalues
    return accuracy
```

4.2. Load the datasets to be evaluated here. Each includes training features ($\mathbf{X}$), and test features ($\mathbf{y}$) for both a low dimensional dataset ($p = 2$ features/predictors) and a higher dimensional dataset ($p = 100$ features/predictors). For each of these datasets there are $n = 1000$ observations of each. They can be found in the data subfolder on github (see link above). Each file is labeled similar to `A2_Q4_X_train_low.csv`, which lets you know whether the dataset is of features, X, targets, y; training or testing; and low or high dimensions.

In [3]:
```
# 4.2
X_train_low = np.loadtxt("A2_Q4_X_train_low.csv", delimiter=",")
y_train_low = np.loadtxt("A2_Q4_y_train_low.csv", delimiter=",")

X_test_low = np.loadtxt("A2_Q4_X_test_low.csv", delimiter=",")
y_test_low = np.loadtxt("A2_Q4_y_test_low.csv", delimiter=",")

X_train_high = np.loadtxt("A2_Q4_X_train_high.csv", delimiter=",")
y_train_high = np.loadtxt("A2_Q4_y_train_high.csv", delimiter=",")

X_test_high = np.loadtxt("A2_Q4_X_test_high.csv", delimiter=",")
y_test_high = np.loadtxt("A2_Q4_y_test_high.csv", delimiter=",")
```

4.3. Train your classifier on first the low dimensional dataset and then the high dimensional dataset with $k = 5$. Evaluate the classification performance on the corresponding test data for each of those trained models. Calculate the time it takes each model to make the predictions and the overall accuracy of those predictions for each corresponding set of test data - state each.

In [5]:
```
# 4.3
import time

knn = Knn()

# train
knn.fit(X_train_low, y_train_low)

# predict
start = time.perf_counter()
y_hat_low = knn.predict(X_test_low, 5)
end = time.perf_counter()

# calculate accuracy and time
print("\nResults for Low Dimensional Data:\n")
```

```
print(f"Accuracy: {accuracy(y_test_low, y_hat_low)}")
print(f"Time: {end-start:.4f}")
```

```
Results for Low Dimensional Data:

Accuracy: 0.925
Time: 0.0685
```

In [6]:
```
# Training with High dimensional data now

# train
knn.fit(X_train_high, y_train_high)

# predict
start = time.perf_counter()
y_hat_high = knn.predict(X_test_high, 5)
end = time.perf_counter()

# calculate accuracy and time
print("\nResults for High Dimensional Data:\n")
print(f"Accuracy: {accuracy(y_test_high, y_hat_high)}")
print(f"Time: {end-start:.4f}")
```

```
Results for High Dimensional Data:

Accuracy: 0.993
Time: 0.1135
```

For the low-dimensional dataset, the classifier achieves an accuracy of 92.5% with relatively fast prediction time. For the high-dimensional dataset, accuracy increases to 99.3%, but prediction time also increases. This reflects the fact that KNN prediction requires computing distances to all training points, and higher dimensionality increases computational cost.

4.4. Compare your implementation's accuracy and computation time to the scikit learn KNeighborsClassifier class. How do the results and speed compare to your implementation? *Hint: your results should be identical to that of the scikit-learn implementation.*

In [15]:
```
#4.4 Using sklern library for knn
from sklearn.neighbors import KNeighborsClassifier
print("\nUsing sklearn KNN Classifier for Low Dimensional Data:\n")
knn = KNeighborsClassifier(n_neighbors=5)
# train
knn.fit(X_train_low, y_train_low)


# predict
start = time.perf_counter()
y_hat_low = knn.predict(X_test_low)
end = time.perf_counter()

# calculate accuracy and time
print(f"Accuracy: {accuracy(y_test_low, y_hat_low)}")
print(f"Time: {end-start:.4f}")

print("\nUsing sklearn KNN Classifier for High Dimensional Data:\n")
knn = KNeighborsClassifier(n_neighbors=5)
# train
knn.fit(X_train_high, y_train_high)
```

```
# predict
start = time.perf_counter()
y_hat_high = knn.predict(X_test_high)
end = time.perf_counter()

# calculate accuracy and time
print(f"Accuracy: {accuracy(y_test_high, y_hat_high)}")
print(f"Time: {end-start:.4f}")
```

```
Using sklearn KNN Classifier for Low Dimensional Data:

Accuracy: 0.925
Time: 0.0035

Using sklearn KNN Classifier for High Dimensional Data:

Accuracy: 0.993
Time: 0.0630
```

The classification accuracy of the custom KNN implementation is identical to that of the scikit-learn KNeighborsClassifier for both low- and high-dimensional datasets, confirming correctness. However, scikit-learn is substantially faster at prediction.

4.5. Some supervised learning algorithms are more computationally intensive during training than testing. What are the drawbacks of the prediction process being slow? In what cases in practice might slow testing (inference) be more problematic than slow training?

Slow prediction (inference) can be problematic because predictions are often required in real time or at scale. If inference is slow, it can lead to poor user experience, system bottlenecks, or inability to deploy models in time-sensitive applications. Slow testing is particularly problematic in applications such as fraud detection, recommendation systems, autonomous driving, medical diagnosis, where decisions must be made immediately. In contrast, slow training is often acceptable because it is done offline and infrequently.

Exercise 5 - Bias-variance tradeoff: exploring the tradeoff with a KNN classifier

[20 points]

This exercise will illustrate the impact of the bias-variance tradeoff on classifier performance by investigating how model flexibility impacts classifier decision boundaries. For this problem, please us Scikit-learn's KNN implementation rather than your own implementation, as you did at the end of the last question.

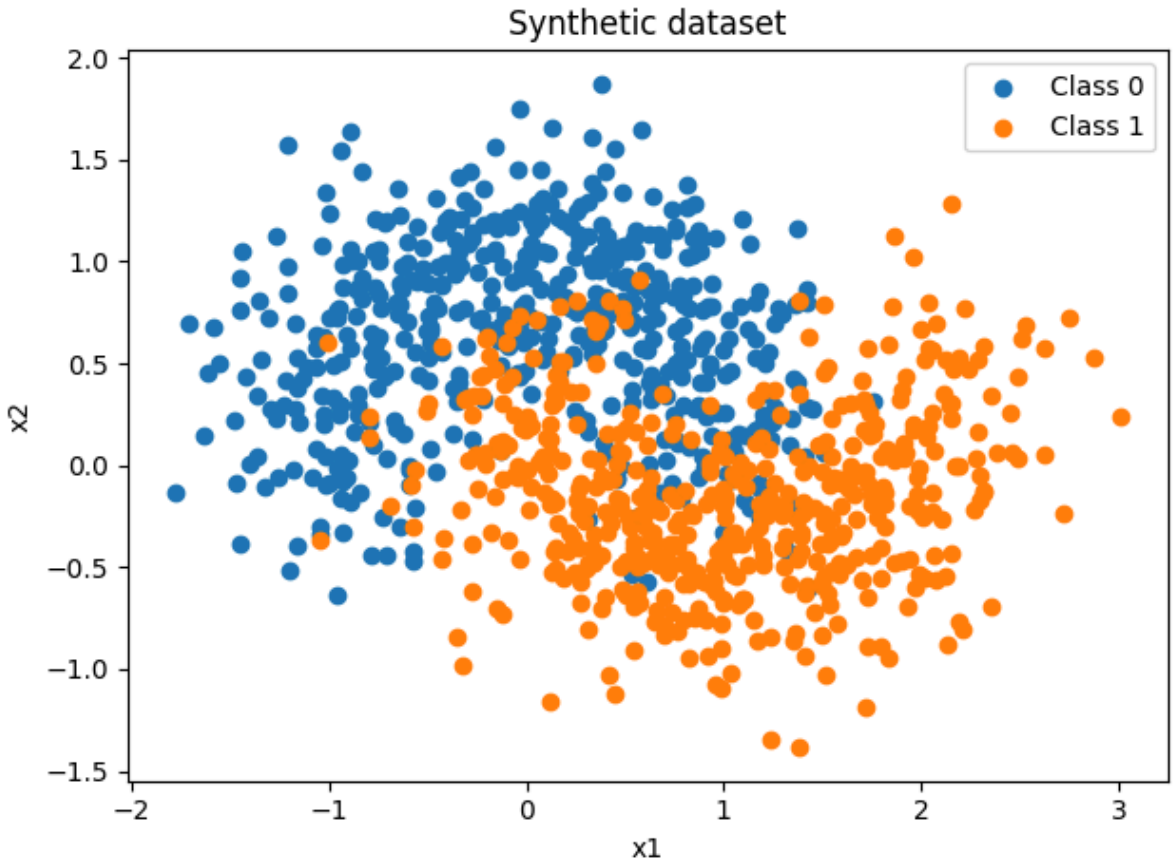
5.1. Create a synthetic dataset (with both features and targets). Use the `make_moons` module with the parameter `noise=0.35` to generate 1000 random samples.

In [11]:
```
# 5.1
import matplotlib.pyplot as plt
from sklearn.datasets import make_moons
```

```
np.random.seed(42)
X, y = make_moons(n_samples=1000, noise=0.35, random_state=42)
```

5.2. Visualize your data: scatterplot your random samples with each class in a different color.

In [12]:
```
# 5.2
plt.figure()
plt.scatter(X[y == 0, 0], X[y == 0, 1], label='Class 0')
plt.scatter(X[y == 1, 0], X[y == 1, 1], label='Class 1')

plt.title("Synthetic dataset")
plt.xlabel("x1")
plt.ylabel("x2")
plt.legend()
plt.tight_layout()
plt.show()
```



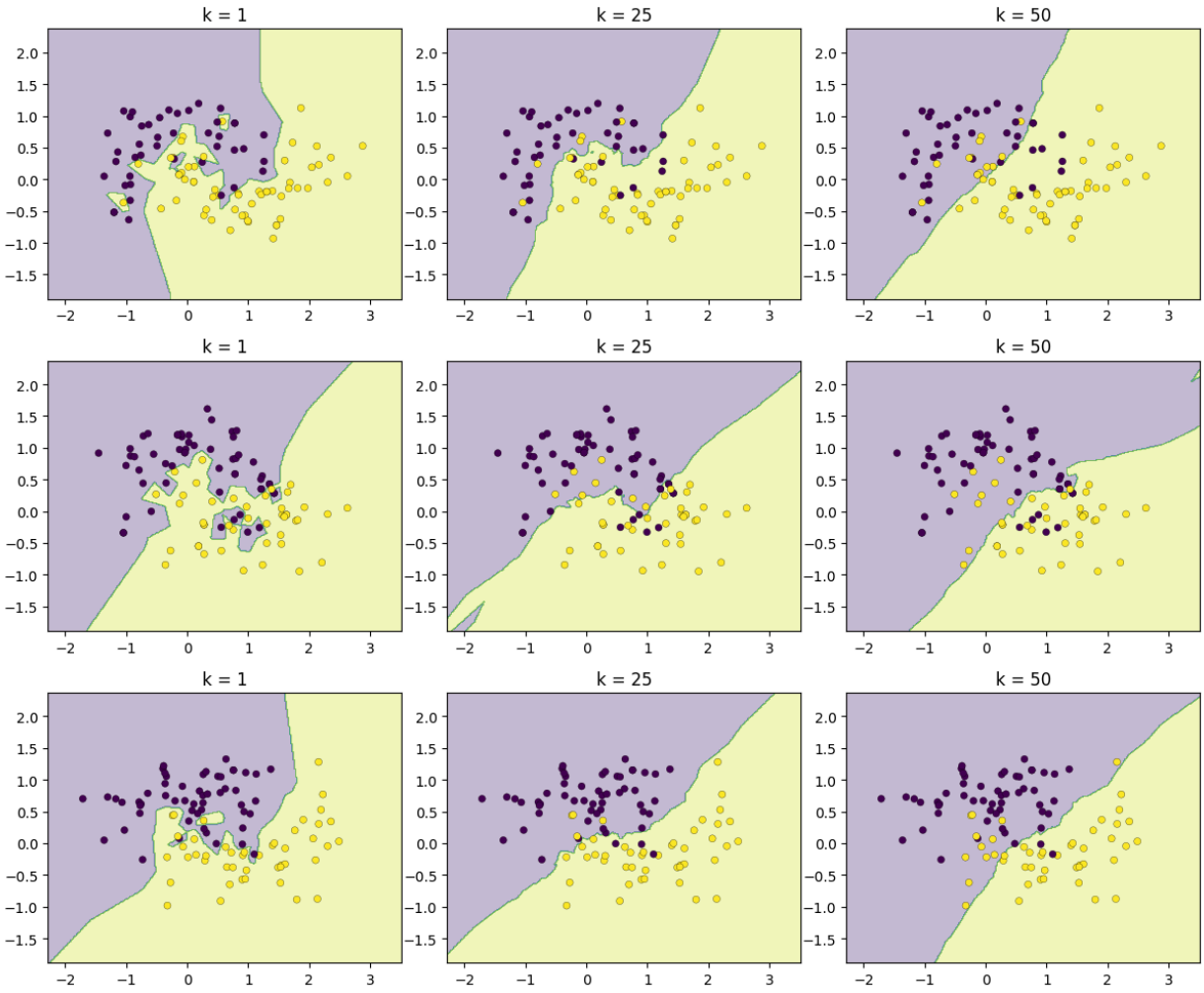
5.3. Create 3 different data subsets by selecting 100 of the 1000 data points at random three times (with replacement). For each of these 100-sample datasets, fit three separate k-Nearest Neighbor classifiers with: $k = \{1, 25, 50\}$. This will result in 9 combinations (3 datasets, each with 3 trained classifiers).

In [13]:
```
# 5.3
rng = np.random.default_rng(42)
subsets = []
for _ in range(3):
    idx = rng.choice(len(X), size=100, replace=True)
    subsets.append((X[idx], y[idx]))
```

```
ks = [1, 25, 50]


def plot_boundary(ax, X_train, y_train, k):
    clf = KNeighborsClassifier(n_neighbors=k)
    clf.fit(X_train, y_train)

    x_min, x_max = X[:, 0].min() - 0.5, X[:, 0].max() + 0.5
    y_min, y_max = X[:, 1].min() - 0.5, X[:, 1].max() + 0.5
    xx, yy = np.meshgrid(
        np.linspace(x_min, x_max, 300),
        np.linspace(y_min, y_max, 300),
    )
    grid = np.c_[xx.ravel(), yy.ravel()]
    Z = clf.predict(grid).reshape(xx.shape)

    ax.contourf(xx, yy, Z, alpha=0.3)
    ax.scatter(
        X_train[:, 0], X_train[:, 1], c=y_train, s=25, edgecolors="k", linewidths
    )
    ax.set_title(f"k = {k}")
    ax.set_xlim(x_min, x_max)
    ax.set_ylim(y_min, y_max)
```

5.4. For each combination of dataset and trained classifier plot the decision boundary (similar in style to Figure 2.15 from *Introduction to Statistical Learning*). This should form a 3-by-3 grid. Each column should represent a different value of k and each row should represent a different dataset.

In [16]:
```
# 5.4
fig, axes = plt.subplots(3, 3, figsize=(12, 10))

for r, (X_sub, y_sub) in enumerate(subsets):
    for c, k in enumerate(ks):
        plot_boundary(axes[r, c], X_sub, y_sub, k)

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



5.5. What do you notice about the difference between the decision boundaries in the rows and the columns in your figure? Which decision boundaries appear to best separate the two classes of data with respect to the training data? Which decision boundaries vary the most as the training data change? Which decision boundaries do you anticipate will generalize best to unseen data and why?

(1) Rows vs. columns:

Within a fixed column (same k), the decision boundaries change across rows because each row is trained on a different sampled dataset. This shows how sensitive the model is to changes in training data. Moving left to right (increasing k), the decision boundaries become progressively smoother. Smaller k produces highly irregular boundaries, while larger k produces smoother, more stable ones.

(2) Best separation on training data: K=1 separates the training data best (it fits the training points most closely).

(3) Varies most when training data change: k=1 varies the most across rows (high sensitivity to the sampled data).

(4) Best generalization to unseen data (and why): k=25 (or similar intermediate k) should generalize best because it is smoother than k=1 (less overfitting) but not as overly smooth as

k=50 (less underfitting).

5.6. Explain the bias-variance tradeoff using the example of the plots you made in this exercise and its implications for training supervised machine learning algorithms.

Small k (e.g., k=1) has low bias but high variance: it fits training data very well but changes a lot with different samples. Large k (e.g., k=50) has high bias but low variance: it is stable but may miss real structure. Choosing an intermediate k balances bias and variance, which usually improves generalization in supervised learning.

Exercise 6 - Bias-variance trade-off II: Quantifying the tradeoff

[18 points]

This exercise explores the impact of the bias-variance tradeoff on classifier performance by looking at the performance on both training and test data.

Here, the value of k determines how flexible our model is.

6.1. Using the function created earlier to generate random samples (using the `make_moons` function setting the `noise` parameter to 0.35), create a new set of 1000 random samples, and call this dataset your test set and the previously created dataset your training set.

In [17]:
```
# 6.1
# Using previously created data as training set
X_train, y_train = X, y

# test set
X_test, y_test = make_moons(n_samples=1000, noise=0.35, random_state=0)
```

6.2. Train a kNN classifier on your training set for $k = 1, 2, \ldots 500$. Apply each of these trained classifiers to both your training dataset and your test dataset and plot the classification error (fraction of incorrect predictions).

In [18]:
```
# 6.2
ks = np.arange(1, 501)

train_err = np.zeros_like(ks, dtype=float)
test_err = np.zeros_like(ks, dtype=float)

for i, k in enumerate(ks):
    knn = KNeighborsClassifier(n_neighbors=k)
    knn.fit(X_train, y_train)

    yhat_train = knn.predict(X_train)
    yhat_test = knn.predict(X_test)

    train_err[i] = np.mean(yhat_train != y_train)
    test_err[i] = np.mean(yhat_test != y_test)
```

```
plt.figure(figsize=(8, 5))
plt.plot(ks, train_err, label="Training error")
plt.plot(ks, test_err, label="Test error")
plt.xlabel("k")
plt.ylabel("Classification error")
plt.title("KNN error vs k")
plt.legend()
plt.show()
```

6.3. What trend do you see in the results?

As k increases, the training error increases, while the test error first decreases and then increases.

6.4. What values of k represent high bias and which represent high variance?

Small values of k correspond to high variance, while large values of k correspond to high bias.

6.5. What is the optimal value of k and why?

The optimal value of k is the one that minimizes the test error, where the bias–variance tradeoff is balanced. As can be seen below for our case the optimal K (where the test error is minimum) comes out to be 179.

In [19]:
```
min_test_error_idx = np.argmin(test_err)
optimal_k = ks[min_test_error_idx]
min_test_error = test_err[min_test_error_idx]

print(f"Optimal k value: {optimal_k}")
print(f"Minimum test error: {min_test_error:.4f}")
```

```
Optimal k value: 179
Minimum test error: 0.1070
```

6.6. In KNN classifiers, the value of k controls the flexibility of the model - what controls the flexibility of other models?

For other models, flexibility is controlled by their respective hyperparameters. For example:

- **Decision Trees:** tree depth or minimum samples per leaf (deeper trees = more flexible)
- **Polynomial Regression:** degree of polynomial (higher degree = more flexible)
- **Ridge/Lasso Regression:** regularization parameter λ (smaller λ = more flexible)
- **Neural Networks:** number of layers/neurons, regularization strength
- **SVM:** kernel choice and regularization parameter C

Exercise 7 - Linear regression and nonlinear transformations

[18 points]

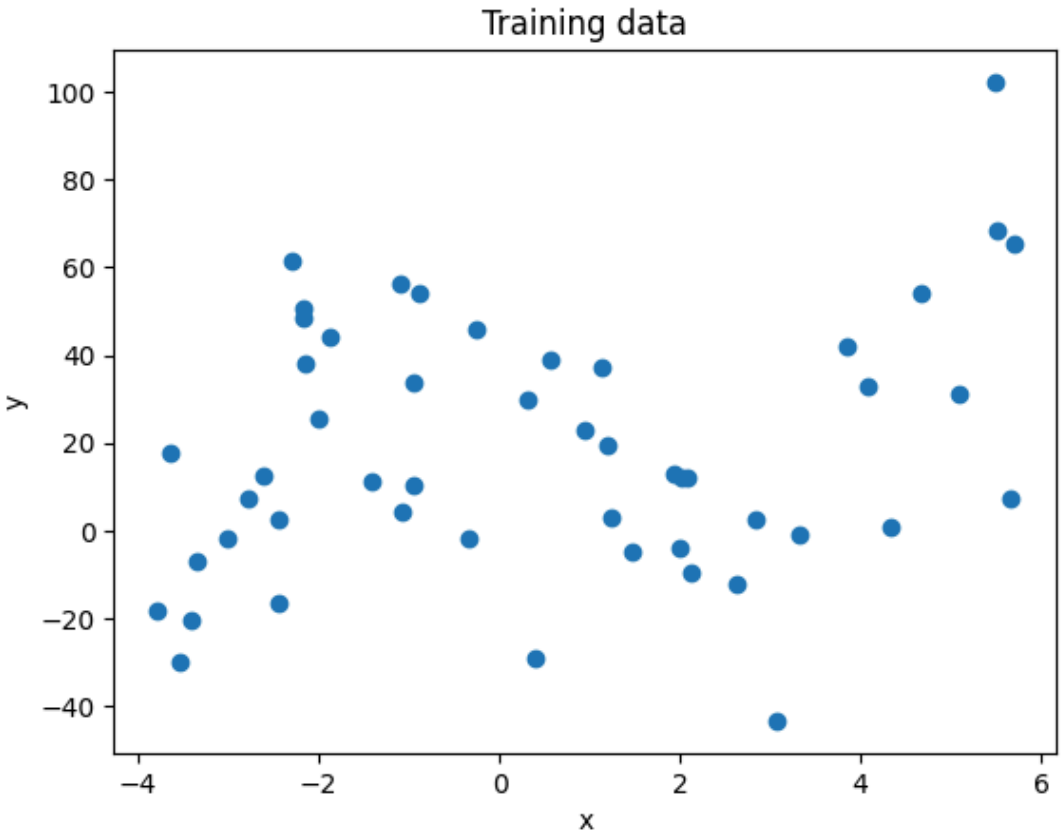
> **Note**
>
> Data for this exercise can be downloaded here

Linear regression can be used to model nonlinear relationships when feature variables are properly transformed to represent the nonlinearities in the data folder. In this exercise, you're given training and test data contained in files "A2_Q7_train.csv" and "A2_Q7_test.csv" in the data. Your goal is to develop a regression algorithm from the training data that performs well on the test data.

Hint: Use the scikit learn LinearRegression module.

7.1. Create a scatter plot of your training data.

In [29]:
```
# 7.1
import numpy as np
import pandas as pd

train = pd.read_csv("A2_Q7_train.csv")
test = pd.read_csv("A2_Q7_test.csv")

x_train = train.x.values
y_train = train.y.values

x_test = test.x.values
y_test = test.y.values

# make plot
plt.scatter(x_train, y_train)
plt.xlabel("x")
plt.ylabel("y")
plt.title("Training data")
plt.show()
```



7.2. Estimate a linear regression model ($y = a_0 + a_1 x$) for the training data and calculate both the R^2 value and mean square error for the fit of that model for the training data. Also provide the equation representing the estimated model (e.g. $y = a_0 + a_1 x$, but with the estimated coefficients inserted. Consider this your baseline model against which you will compare other model options. *Evaluating performance on the training data is not a measure of how well this model would generalize to unseen data. We will evaluate performance on the test data once we see our models fit the training data decently well.*

In [30]:
```
# 7.2
from sklearn.linear_model import LinearRegression
from sklearn.metrics import r2_score, mean_squared_error

# model
lin = LinearRegression()
lin.fit(x_train.reshape(-1, 1), y_train)
y_hat_train = lin.predict(x_train.reshape(-1, 1))

# param
a0 = lin.intercept_
a1 = lin.coef_[0]

# eval
r2_train_lin = r2_score(y_train, y_hat_train)
mse_train_lin = mean_squared_error(y_train, y_hat_train)

# print
print(f"Linear model: y = {a0:.3f} + {a1:.3f} x")
print("Train R^2:", r2_train_lin)
print("Train MSE:", mse_train_lin)
```

```
Linear model: y = 17.205 + 2.591 x
Train R^2: 0.06486123304769698
Train MSE: 791.4167471701106
```

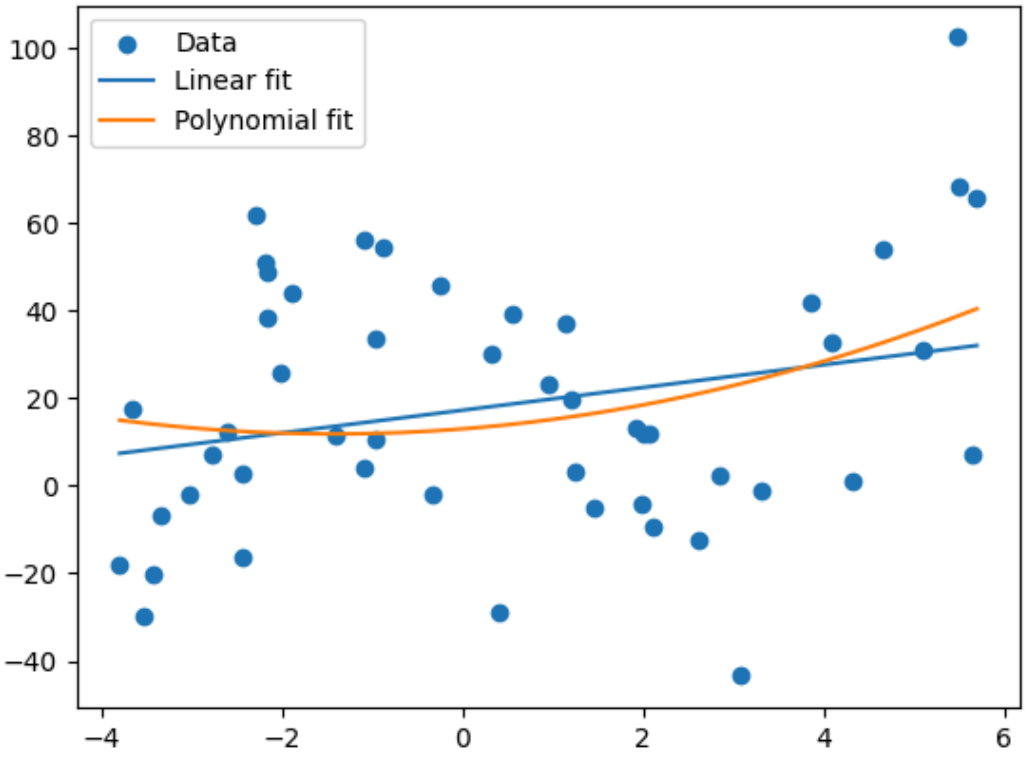
7.3. If features can be nonlinearly transformed, a linear model may incorporate those non-linear feature transformation relationships in the training process. From looking at the scatter plot of the training data, choose a transformation of the predictor variable, x that may make sense for these data. This will be a multiple regression model of the form $y = a_0 + a_1z_1 + a_2z_2 + \ldots + a_nz_n$. Here z_i could be any transformations of x - perhaps it's $\frac{1}{x}$, $log(x)$, $sin(x)$, x^k (where k is any power of your choosing). Provide the estimated equation for this multiple regression model (e.g. if you chose your predictors to be $z_1 = x$ and $z_2 = log(x)$, your model would be of the form $y = a_0 + a_1x + a_2log(x)$. Also provide the R^2 and mean square error of the fit for the training data.

In [48]:
```
# 7.3
x_train_poly = np.column_stack([x_train, x_train**2])

# model
poly = LinearRegression()
poly.fit(x_train_poly, y_train)
y_hat_ploy_train = poly.predict(x_train_poly)

# param
a0_p = lin.intercept_
a1_p, a2_p = poly.coef_

# eval
r2_train_poly = r2_score(y_train, y_hat_ploy_train)
mse_train_poly = mean_squared_error(y_train, y_hat_ploy_train)

# print
print(f"Polynomial model: y = {a0_p:.3f} + {a1_p:.3f} x + {a2_p:.3f} x^2")
print("Train R^2:", r2_train_poly)
print("Train MSE:", mse_train_poly)
```

```
Polynomial model: y = 17.205 + 1.606 x + 0.562 x^2
Train R^2: 0.08529040403484756
Train MSE: 774.1273473276406
```

7.4. Visualize the model fit to the training data. Using both of the models you created in parts (b) and (c), plot the original data (as a scatter plot) AND the curves representing your models (each as a separate curve) from (b) and (c).

In [49]:
```
# 7.4
x_grid = np.linspace(x_train.min(), x_train.max(), 300)
y_lin_curve = lin.predict(x_grid.reshape(-1, 1))
y_poly_curve = poly.predict(np.column_stack([x_grid, x_grid**2]))

plt.scatter(x_train, y_train, label="Data")
plt.plot(x_grid, y_lin_curve, label="Linear fit")
plt.plot(x_grid, y_poly_curve, label="Polynomial fit")
plt.legend()
plt.show()
```



7.5. Now its time to compare your models and evaluate the generalization performance on held out test data. Using the models above from (b) an (c), apply them to the test data and estimate the R^2 and mean square error of the test dataset.

In [50]:
```
# 7.5
# linear
y_hat_test_lin = lin.predict(x_test.reshape(-1, 1))
print("Linear test R^2:", r2_score(y_test, y_hat_test_lin))
print("Linear test MSE:", mean_squared_error(y_test, y_hat_test_lin))

# poly
y_hat_test_poly = poly.predict(np.column_stack([x_test, x_test**2]))
print("Poly test R^2:", r2_score(y_test, y_hat_test_poly))
print("Poly test MSE:", mean_squared_error(y_test, y_hat_test_poly))
```

```
Linear test R^2: -0.132899284725984
Linear test MSE: 1116.6632365616088
Poly test R^2: -0.1202562792666435
Poly test MSE: 1104.2014232420709
```

7.6. Which models perform better on the training data, and which on the test data? Why?

The polynomial model performs slightly better than the linear model on the test data, as it has a lower MSE and higher R^2. However, both models perform poorly on the test data, indicating limited generalization.

7.7. Imagine that the test data were significantly different from the training dataset. How might this affect the predictive capability of your model? How would the accuracy of generalization performance be impacted? Why?

If the test data do not follow the same distribution as the training data, the model struggles because it is asked to make predictions in situations it was not trained on.

BONUS Question

In [74]:
```
from ucimlrepo import fetch_ucirepo

# fetch dataset
adult = fetch_ucirepo(id=2)

# data (as pandas dataframes)
X = adult.data.features
y = adult.data.targets

# combine X and y for easier handling
data = pd.concat([X, y], axis=1)
```

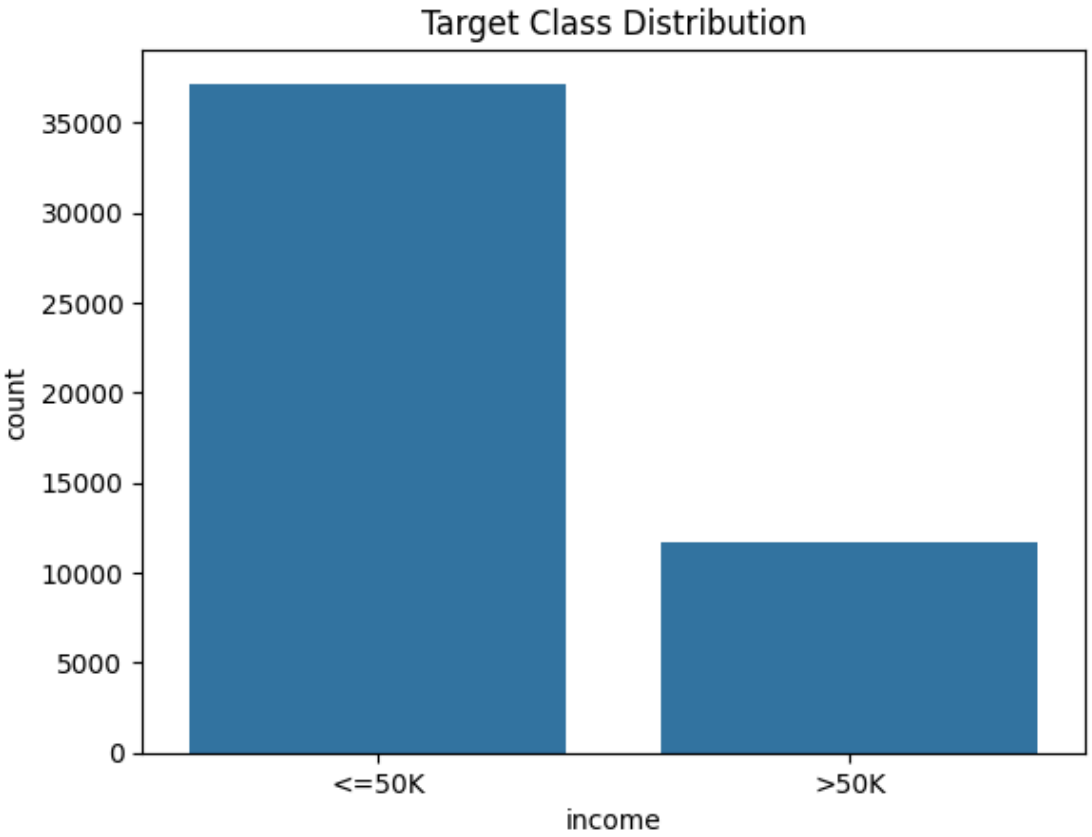
Let's explore this data set first:

In [75]:
```
print("Unique target values:", data['income'].unique())
print("\nTarget distribution:")
print(data['income'].value_counts())
```

```
Unique target values: <StringArray>
['<=50K', '>50K', '<=50K.', '>50K.']
Length: 4, dtype: str

Target distribution:
income
<=50K     24720
<=50K.    12435
>50K       7841
>50K.      3846
Name: count, dtype: int64
```

As we can see there are dulicates of each target because there's a period at the end of them. Let's clean that up.

In [76]:
```
import seaborn as sns

data["income"] = data["income"].str.strip().str.rstrip('.')

sns.countplot(x=data["income"])
plt.title("Target Class Distribution")
plt.show()
```



There's a class imbalance as most people earn ≤50K, and fewer earn >50K. We'll keep this in mind while evaluating our model

In [77]:
```
print("No. of rows and columns in feature data:", data.shape)
data.head()
```

```
No. of rows and columns in feature data: (48842, 15)
```

Out[77]:

	age	workclass	fnlwgt	education	education-num	marital-status	occupation	relationship	rac
0	39	State-gov	77516	Bachelors	13	Never-married	Adm-clerical	Not-in-family	Whit
1	50	Self-emp-not-inc	83311	Bachelors	13	Married-civ-spouse	Exec-managerial	Husband	Whit
2	38	Private	215646	HS-grad	9	Divorced	Handlers-cleaners	Not-in-family	Whit
3	53	Private	234721	11th	7	Married-civ-spouse	Handlers-cleaners	Husband	Blac
4	28	Private	338409	Bachelors	13	Married-civ-spouse	Prof-specialty	Wife	Blac

Check for NANs

In [78]:
```
# Check for NaNs in each column
nan_counts = data.isna().sum()
print(nan_counts)
```

```
age                0
workclass        963
fnlwgt             0
education          0
education-num      0
marital-status     0
occupation       966
relationship       0
race               0
sex                0
capital-gain       0
capital-loss       0
hours-per-week     0
native-country   274
income             0
dtype: int64
```

Since there are 48k rows, let's remove the rows having Null values

In [79]:
```
data= data.dropna().reset_index(drop=True)

print("No. of rows and columns after removing NaNs:", data.shape)
```

```
No. of rows and columns after removing NaNs: (47621, 15)
```

Handling categorical values and Scaling Numeric Ones

In [80]:
```
#data
X= data.drop(columns=['income'])
#target
y= data['income'].map({'<=50K': 0, '>50K': 1})
```

In [81]:
```
from sklearn.preprocessing import StandardScaler

# One-hot encode categorical features
categorical_cols = [
    'workclass', 'education', 'marital-status',
    'occupation', 'relationship', 'race', 'sex', 'native-country'
]

X_encoded = pd.get_dummies(X, columns=categorical_cols, drop_first=True, dtype=ir
numeric_cols = ['age', 'fnlwgt', 'education-num', 'capital-gain', 'capital-loss',

scaler = StandardScaler()
X_encoded[numeric_cols] = scaler.fit_transform(X_encoded[numeric_cols])


X_encoded
```

Out[81]:

	age	fnlwgt	education-num	capital-gain	capital-loss	hours-per-week	workclass_Feder
0	0.026501	-1.062924	1.132729	0.144629	-0.217456	-0.048943	
1	0.837781	-1.008031	1.132729	-0.145735	-0.217456	-2.251188	
2	-0.047252	0.245517	-0.424726	-0.145735	-0.217456	-0.048943	
3	1.059039	0.426206	-1.203454	-0.145735	-0.217456	-0.048943	
4	-0.784780	1.408394	1.132729	-0.145735	-0.217456	-0.048943	
...	...	...	...	...	...	...	
47616	-0.416016	0.525573	1.132729	-0.145735	-0.217456	-0.048943	
47617	0.026501	0.243367	1.132729	-0.145735	-0.217456	-0.375201	
47618	-0.047252	1.754843	1.132729	-0.145735	-0.217456	0.766703	
47619	0.395264	-1.002537	1.132729	0.582847	-0.217456	-0.048943	
47620	-0.268510	-0.071794	1.132729	-0.145735	-0.217456	1.582350	

47621 rows × 100 columns

Let's split our data

In [82]:
```
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(
    X_encoded, y, test_size=0.2, random_state=42, stratify=y
) # since our classes are imbalanced, we use stratify to maintain the same distri

print("Training set shape:", X_train.shape)
print("Test set shape:", X_test.shape)
```

```
Training set shape: (38096, 100)
Test set shape: (9525, 100)
```

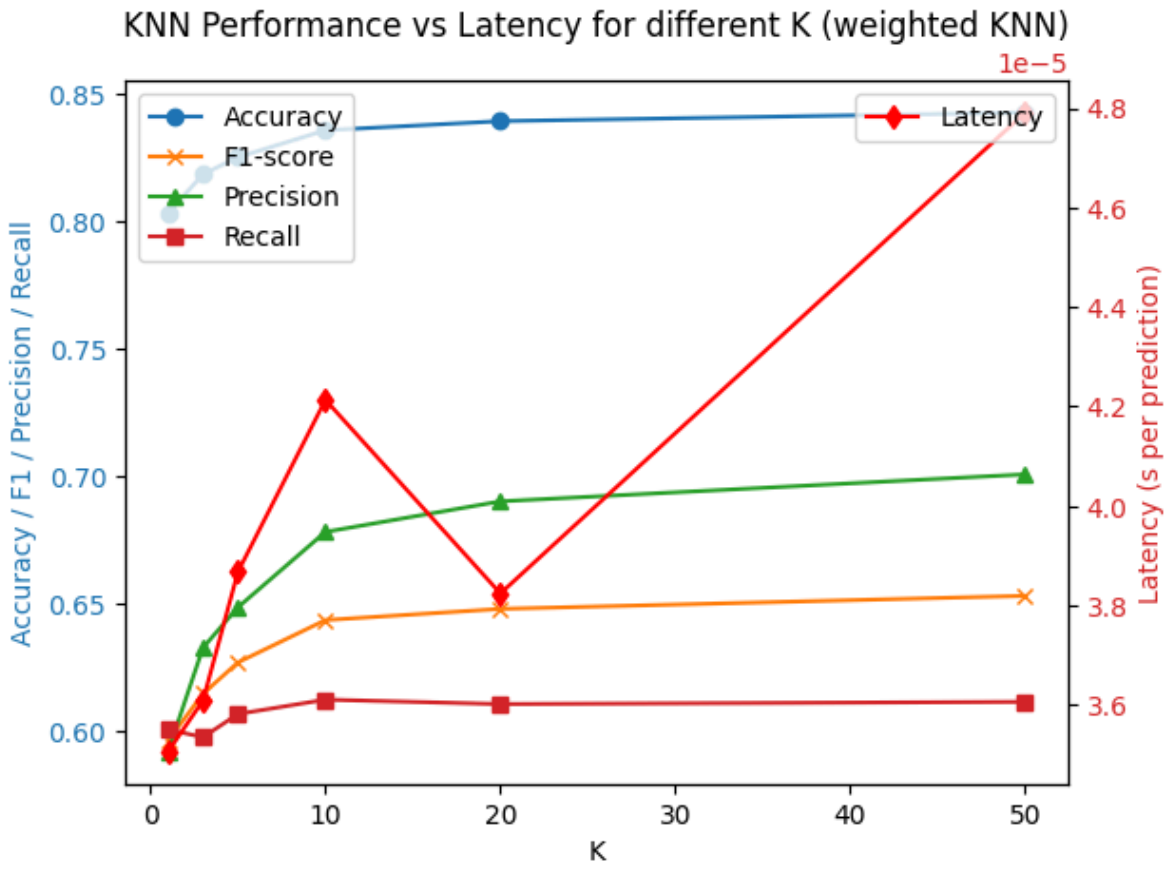
Let's start our Training process

In [85]:
```
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import accuracy_score, f1_score, precision_score, recall_sco

# Function to evaluate kNN on imbalanced dataset

def evaluate_knn(k, X_train, X_test, y_train, y_test):
    model = KNeighborsClassifier(n_neighbors=k, weights='distance') # since class
    model.fit(X_train, y_train)

    start = time.time()
    y_pred = model.predict(X_test)
    end = time.time()

    latency = (end - start) / len(X_test)
    accuracy = accuracy_score(y_test, y_pred)
    f1 = f1_score(y_test, y_pred)
```

```
    precision = precision_score(y_test, y_pred)
    recall = recall_score(y_test, y_pred)

    return latency, accuracy, f1, precision, recall
```

Varying K

In [86]:
```
results_k = []
for k in [1, 3, 5, 10, 20, 50]:
    latency, acc, f1, prec, rec = evaluate_knn(k, X_train, X_test, y_train, y_tes
    results_k.append({
        'K': k, 'latency': latency, 'accuracy': acc,
        'f1': f1, 'precision': prec, 'recall': rec
    })

df_k = pd.DataFrame(results_k)

# Plot K vs Performance and Latency
fig, ax1 = plt.subplots()
ax1.set_xlabel('K')
ax1.set_ylabel('Accuracy / F1 / Precision / Recall', color='tab:blue')
ax1.plot(df_k['K'], df_k['accuracy'], label='Accuracy', marker='o')
ax1.plot(df_k['K'], df_k['f1'], label='F1-score', marker='x')
ax1.plot(df_k['K'], df_k['precision'], label='Precision', marker='^')
ax1.plot(df_k['K'], df_k['recall'], label='Recall', marker='s')
ax1.tick_params(axis='y', labelcolor='tab:blue')
ax1.legend(loc='upper left')

ax2 = ax1.twinx()
ax2.set_ylabel('Latency (s per prediction)', color='tab:red')
ax2.plot(df_k['K'], df_k['latency'], color='red', marker='d', label='Latency')
ax2.tick_params(axis='y', labelcolor='tab:red')
ax2.legend(loc='upper right')
plt.title("KNN Performance vs Latency for different K (weighted KNN)")
plt.show()
```



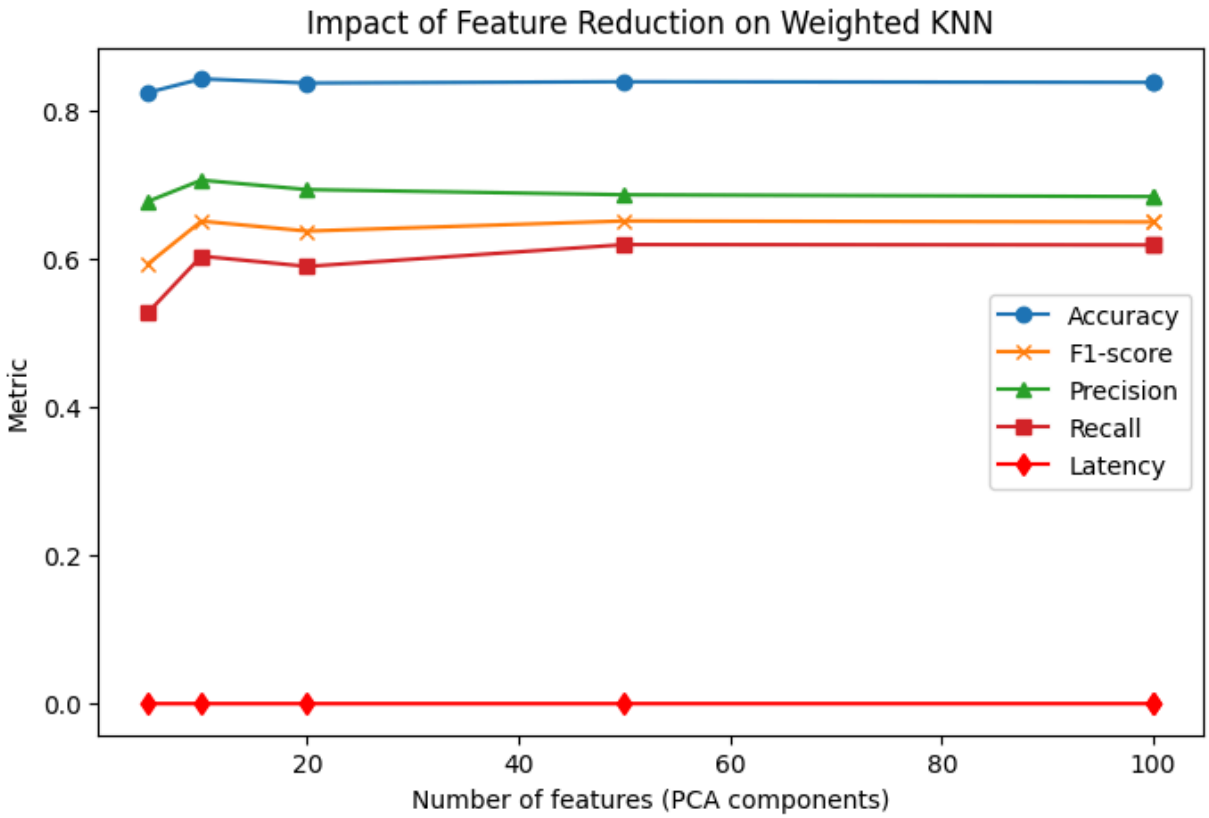
The above gaph depicts the performance dynamics of a Weighted K-Nearest Neighbors (KNN) algorithm, highlighting a clear trade-off between predictive quality and computational efficiency as the parameter K scales.

Initially, as k increases from 1 to 10, there is a sharp upward trend across Accuracy, Precision, and F1-score, indicating that the model benefits from a larger consensus of neighbors to smooth out noise. However, after $K = 20$, these metrics reach a plateau, with Accuracy stabilizing near 0.84 and Precision showing a slight decline, signaling diminishing returns. In contrast, the Latency increases almost linearly after $K = 5$, reaching its peak at $K = 50$ due to the increased computational burden of processing more neighbors. Notably, the Recall remains consistently lower than other metrics, hovering around 0.60, which suggests that while the model is generally accurate, it struggles to identify all positive instances, which is a common sign of class imbalance. Ultimately, the "sweet spot" for this model lies between $K = 10$ and $K = 20$, where performance is maximized before the rising latency becomes a significant bottleneck.

Let's vary our number of features by using PCA

In [ ]:
```
from sklearn.decomposition import PCA

results_pca = []
for n_features in [5, 10, 20, 50, 100, X_encoded.shape[1]]:
    pca = PCA(n_components=n_features)
    X_train_pca = pca.fit_transform(X_train)
    X_test_pca = pca.transform(X_test)
```

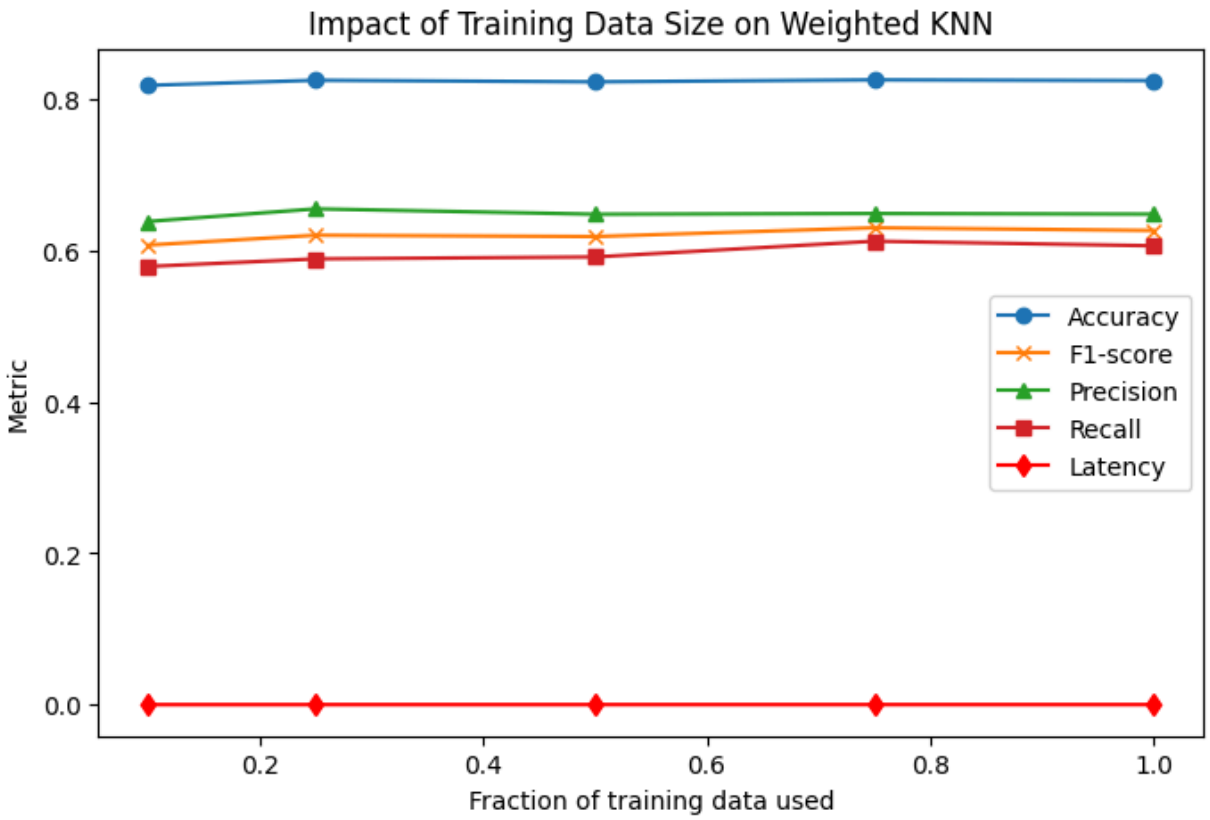
```
    latency, acc, f1, prec, rec = evaluate_knn(15, X_train_pca, X_test_pca, y_tra
    results_pca.append({
        'features': n_features, 'latency': latency, 'accuracy': acc,
        'f1': f1, 'precision': prec, 'recall': rec
    })

df_pca = pd.DataFrame(results_pca)

plt.figure(figsize=(8,5))
plt.plot(df_pca['features'], df_pca['accuracy'], label='Accuracy', marker='o')
plt.plot(df_pca['features'], df_pca['f1'], label='F1-score', marker='x')
plt.plot(df_pca['features'], df_pca['precision'], label='Precision', marker='^')
plt.plot(df_pca['features'], df_pca['recall'], label='Recall', marker='s')
plt.plot(df_pca['features'], df_pca['latency'], label='Latency', marker='d', colo
plt.xlabel('Number of features (PCA components)')
plt.ylabel('Metric')
plt.title('Impact of Feature Reduction on Weighted KNN')
plt.legend()
plt.show()
```



The feature reduction analysis reveals that the Weighted KNN model reaches peak performance with as few as 10 PCA components. Metrics such as Accuracy and Precision plateau almost immediately, indicating that the first 10 components capture nearly all relevant variance for classification. Latency remains negligible and stable across the tested range, while the consistently lower Recall (approx 0.60) confirms that the model's primary weakness is a structural sensitivity issue (likely due to class imbalance) rather than high-dimensional noise.

Varying training data size

In [91]:
```
results_train_size = []
for frac in [0.1, 0.25, 0.5, 0.75, 1.0]:
    X_train_sub = X_train[:int(len(X_train)*frac)]
    y_train_sub = y_train[:int(len(y_train)*frac)]

    latency, acc, f1, prec, rec = evaluate_knn(5, X_train_sub, X_test, y_train_su
    results_train_size.append({
        'train_frac': frac, 'latency': latency, 'accuracy': acc,
        'f1': f1, 'precision': prec, 'recall': rec
    })

df_train = pd.DataFrame(results_train_size)

plt.figure(figsize=(8,5))
plt.plot(df_train['train_frac'], df_train['accuracy'], label='Accuracy', marker='
plt.plot(df_train['train_frac'], df_train['f1'], label='F1-score', marker='x')
plt.plot(df_train['train_frac'], df_train['precision'], label='Precision', marker
plt.plot(df_train['train_frac'], df_train['recall'], label='Recall', marker='s')
plt.plot(df_train['train_frac'], df_train['latency'], label='Latency', marker='d'
plt.xlabel('Fraction of training data used')
plt.ylabel('Metric')
plt.title('Impact of Training Data Size on Weighted KNN')
plt.legend()
plt.show()
```



The size of the training set (N) is a critical factor in KNN latency because the algorithm must calculate distances to every training point for each new prediction, resulting in a computational complexity of O(N). Our analysis of the impact of training data size revealed significant redundancy, as model performance metrics like Accuracy and Recall remained remarkably stable even when using as little as 10% to 25% of the total available data. Even

though the plot shows a flat line for latency, in a real-world production environment with millions of rows, that line would eventually curve upward.

In large-scale deployments, utilizing a smaller but representative training set is significantly faster than processing a massive one, directly reducing the "prediction time" bottleneck identified as vital for real-world applications. Consequently, the decision to use a truncated training set allows the system to achieve "instant" predictions and minimal latency without sacrificing the model's overall Accuracy or Precision.

Final Recommendation

Based on the data, my final recommendation for a real-time system is a Weighted KNN model using $K = 15$ and 10 PCA components.
My analysis of the K value showed that while accuracy and F1-score peak around $K = 10$, latency starts to climb significantly as you add more neighbors, making $K = 15$ a perfect middle ground for stability and speed. To further slash latency, I found that reducing the data to just 10 PCA components maintains an accuracy of ≈ 0.84 while removing 90% of the feature noise that would otherwise slow down distance calculations.

Finally, since the model's performance was almost identical whether using 10% or 100% of the training data, I recommend using a smaller, representative training subset to keep the $O(N)$ search time as low as possible. This configuration ensures the "instant" response times needed for applications like self-driving cars or web recommendations without compromising on predictive power.

In [ ]: