ASL Sheet 4: Overfitting and Underfitting

Exercise 1: Overfitting and Underfitting: Regression

We want to investigate overfitting and underfitting in a regression example. Consider the hypothesis spaces $H_i$defined by:

$$H_i = \left\{ f(x) = \sum_{k=0}^{i} \theta_k x^k | \theta \in \mathbb{R}^{\hat{i}+1} \right\},$$

So, for example, H_0 yields a constant model while H_2 yields the space:

$$H_2 = \{f(x) = \theta_0 + \theta_1 x + \theta_2 x^2 | \theta_0, \theta_1, \theta_2 \in \mathbb{R}\}.$$

a) Consider the file "*regr_train.csv*" which contains two variables x and y with 50 observations each. Fit models to the training data using the hypothesis spaces $H_1, H_2, \dots, H_{12}$ and the quadratic loss function. Visualize the training data and the fitted models.

After downloading the data and compiling the following Figure 1:

```
degrees = range(1, 13)
models = []

plt.figure(figsize=(18, 12))
for i, d in enumerate(degrees, 1):
    poly = PolynomialFeatures(degree=d, include_bias=True)
    X_poly = poly.fit_transform(x)

    model = LinearRegression()
    model.fit(X_poly, y)
    y_pred = model.predict(X_poly)
    models.append((d, model, poly, y_pred))

    # Plot
    plt.subplot(3, 4, i)
    plt.scatter(x, y, label="Training data", color='black')
    plt.plot(np.sort(x, axis=0), model.predict(poly.transform(np.sort(x, axis=0))), color='red')
    plt.title(f"Degree {d}")
    plt.xlabel("x")
    plt.ylabel("y")
    plt.grid(True)

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

Figure 1: Fitting and visualizing models with training data code

We can observe these 12 plots on Figure 2, Figure 3 and Figure 4 on how the model fits depending on the degrees.

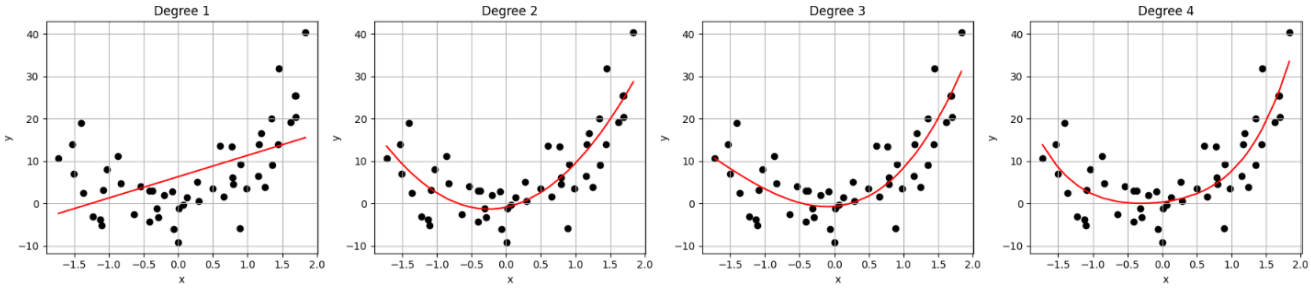


Figure 2: 1 - 4 Degrees training data plots

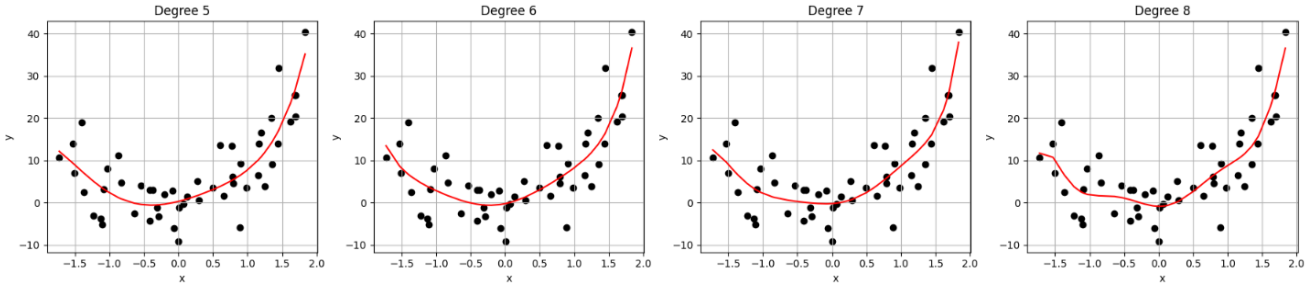


Figure 3: 2 - 8 Degrees training data plots

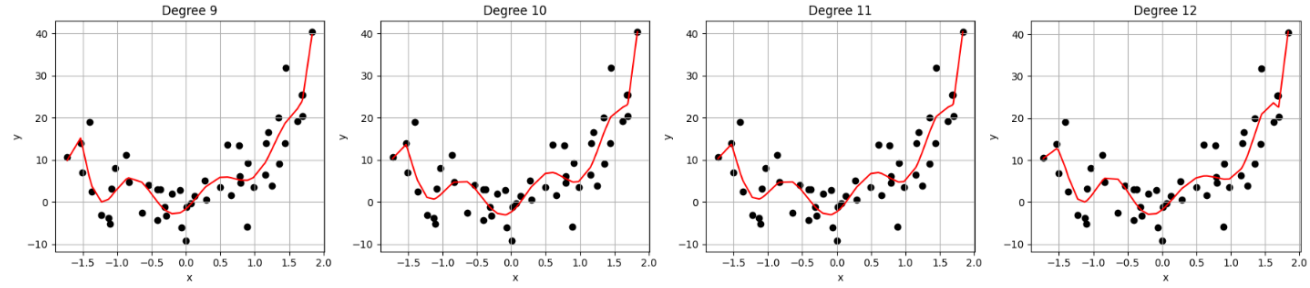


Figure 4: 9 - 12 Degrees training data plots

b) Discuss your models and plots from a). Which hypothesis space do you consider most appropriate here? Why?

From the plots for polynomial degrees 1 through 12, we observe clear signs of both **underfitting** and **overfitting** depending on the degrees taken. We are explaining this with three different "categories" where we can classify our models:

- Low-degree models: mostly the first 2 models.
  - They are too simple and fail to capture the curvature of the data. Therefore, they are producing a high bias and **underfitting** the training data.
- High-degree models: we can consider here the last four models (Figure 4).
  - These models fit the training data extremely closely, capturing even minor fluctuations. So, the curves show excessive oscillation, especially near the edges of the domain.
  - This is what we call **overfitting**, meaning that there's a low training error but likely poor generalization to new data.
- Moderate-degree models: models 4, 5 and 6.
  - These models are capturing the general pattern in the data without overreacting to noise (outliers).
  - The curve is smooth (mostly in 4 and 5) and aligns well with the distribution of training points. These models strike a good balance between bias and variance.

To summarize the information given, based on the visual evidence, we are considering the **hypothesis space H_4** that we can see on Figure 5 (could also be H_5) most appropriate. It captures the underlying structure of the data well while avoiding the overfitting behaviours. This suggests it is likely to generalize better to unseen data.

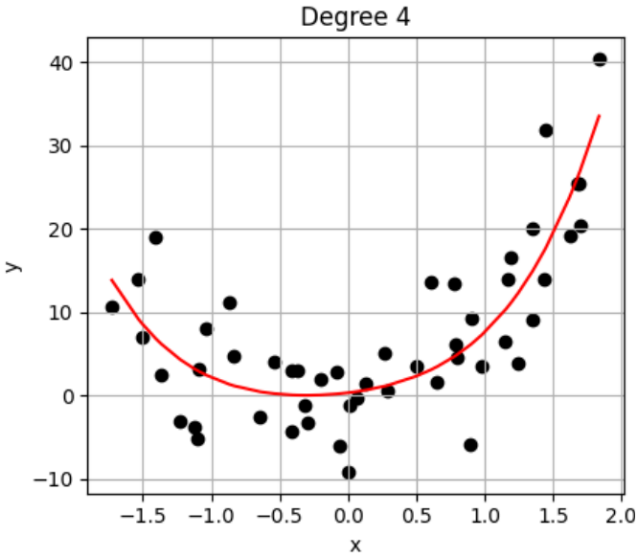


Figure 5: Chose hypothesis space (H4)

c) Now we want to compare the models' performances on unseen test data. Consider the second file "*regr_test.csv*". Visualize the test data together with the models fitted on the training data.

After running the code from Figure 6.

```
plt.figure(figsize=(18, 12))
for i, (d, model, poly, _) in enumerate(models, 1):
    plt.subplot(3, 4, i)
    plt.scatter(x_test, y_test, label='Test data', color='blue')
    x_sorted = np.sort(x, axis=0)
    plt.plot(x_sorted, model.predict(poly.transform(x_sorted)), color='red', label='Fitted model')
    plt.title(f"Degree {d}")
    plt.xlabel("x")
    plt.ylabel("y")
    plt.grid(True)
    plt.legend()

plt.tight_layout()
plt.suptitle("Test Data vs Fitted Models", y=1.02, fontsize=16)
plt.show()
```

Figure 6: Test data performance

We can visualize how the test data performance together with the 12 different models fitted with the training data on Figure 7, Figure 8 and Figure 9.

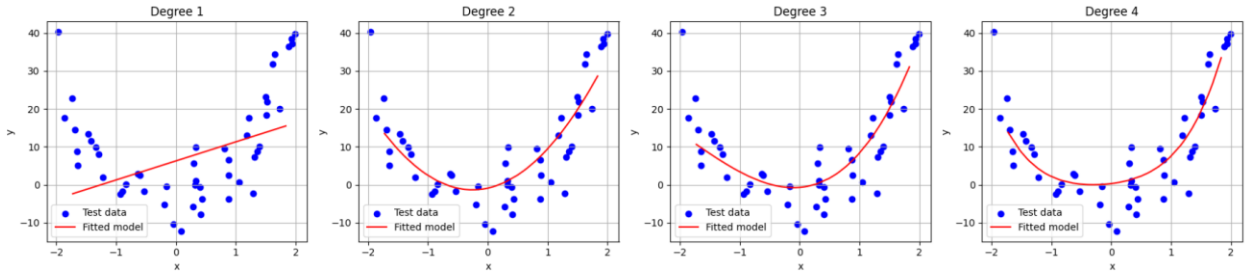


Figure 7: 1 - 4 Degrees test data plots

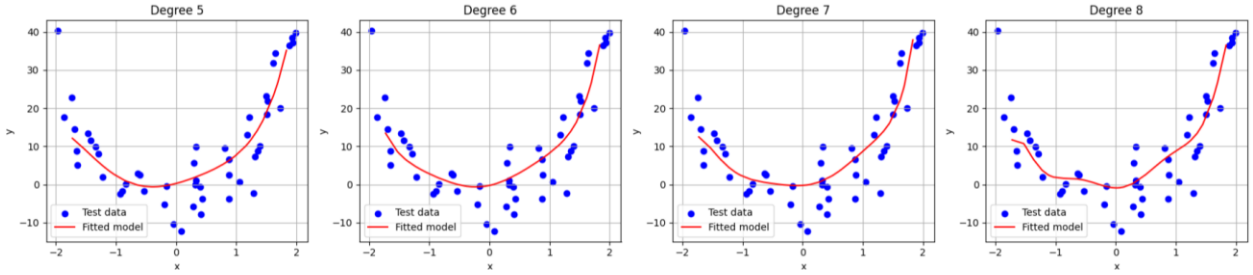


Figure 8: 5 - 8 Degrees test data plots

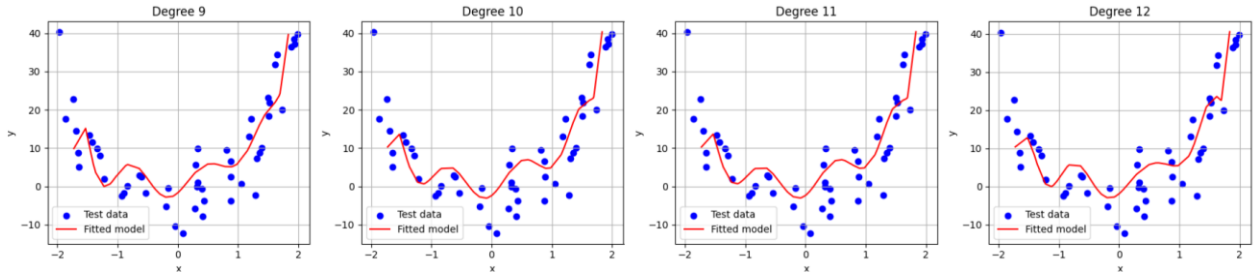


Figure 9: 9 - 12 Degrees test data plots

d) Calculate and visualize the test errors between the test observations and their predictions for all fitted models and comment on the results.

We compile the following code Figure 10:

```python
# Compute test MSEs
test_errors = []
for d, model, poly, _ in models:
    X_test_poly = poly.transform(x_test)
    y_pred_test = model.predict(X_test_poly)
    mse = mean_squared_error(y_test, y_pred_test)
    test_errors.append((d, mse))

# Convert to DataFrame
df_errors = pd.DataFrame(test_errors, columns=["Degree", "Test MSE"])

# Plot test MSE vs degree
plt.figure(figsize=(8, 5))
plt.plot(df_errors["Degree"], df_errors["Test MSE"], marker='o')
plt.title("Test MSE vs Polynomial Degree")
plt.xlabel("Polynomial Degree")
plt.ylabel("Mean Squared Error")
plt.grid(True)
plt.xticks(df_errors["Degree"])
plt.show()

# Display error table
print(df_errors)
```

Figure 10: MSE between test and predictions visualization code

Obtaining the graph Figure 11.

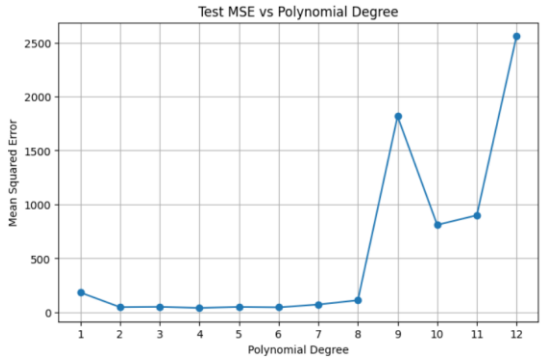


Figure 11: MSE vs Polynomial Degree Plot

We can see in Figure 12 the values of the MSE.

	Degree	Test MSE
0	1	183.641401
1	2	47.951063
2	3	50.863013
3	4	40.534139
4	5	49.908806
5	6	45.905565
6	7	72.121504
7	8	111.779997
8	9	1816.930268
9	10	810.445485
10	11	899.452555
11	12	2559.824403

Figure 12: MSE of each Polynomial Degree Table

These test errors across polynomial degrees show a clear pattern of underfitting and overfitting:

- Low-degree model is the one with 1 degree, exhibit a high-test error (MSE ≈ 183.64), that is indicating there's underfitting.
- Moderate-degree models from 2 to 6 degrees, they achieve low and stable test errors; MSE between 40 and 50.
    - As we visually considered before, **degree 4** achieves the lowest test MSE (MSE ≈ 40.53), suggesting that this model is generalizing the best performance.
- High-degree models from 9 to 12 degrees, show a huge rise in test error, with degree 12 reaching an MSE of ≈ 2559.82. This is a strong sign of overfitting, as we indicated before, the model is fitting the training outliers, and this leads to poor performance on unseen data.

Summarizing the information above, the best generalization is achieved by polynomial degrees 3 to 5, with **degree 4** being the most optimal. These models balances complexity and generalization while they avoid the high variance seen in overfitted higher-degree models. This is great because it matches and confirms the visual analysis we did in parts a) and b).

Homework 4

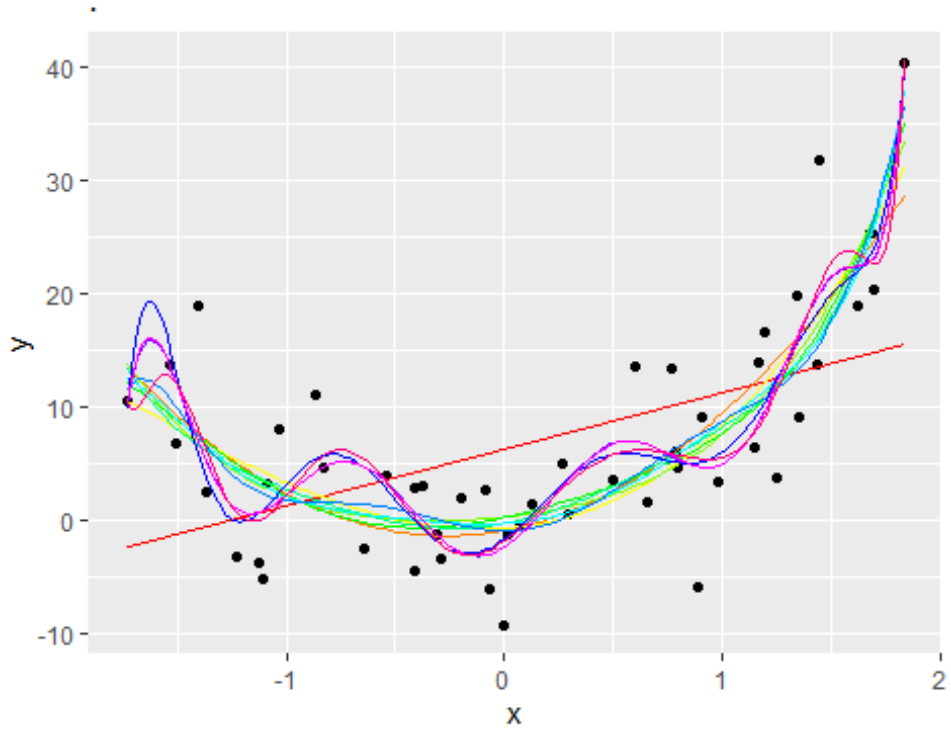
1

```
library(ggplot2)
data <- read.csv("regr_train.csv")
head(data)
```

```
            x         y
1 -1.53147767 13.861916
2 -0.06388107 -6.101648
3  0.60482811 13.642275
4 -1.72641329 10.572149
5 -0.53880879  3.955313
6 -1.10438518 -5.169619
```

A

```
models <- list()

for (i in 1:12) {
  formula <- as.formula(paste("y ~ poly(x,", i, ", raw=TRUE)"))
  models[[i]] <- lm(formula, data = data)
}

x_grid <- seq(min(data$x), max(data$x), length.out = 200)
plot_data <- data.frame(x = x_grid)

gg <- ggplot(data, aes(x, y)) +
  geom_point(color = "black") +
  ggtitle(".")

for (i in 1:12) {
  pred <- predict(models[[i]], newdata = plot_data)
  gg <- gg + geom_line(data = data.frame(x = x_grid, y = pred),
                       aes(x = x, y = y),
                       color = rainbow(12)[i],
                       linetype = "solid")
}

print(gg)
```



B

By analyzing the models and the plot from part (a), we can clearly see that the degree 1 model (H_1) fails to capture the curved trend in the data this is a clear case of underfitting.

On the other hand, high-degree models (such as $H_{1\ 1}$ or $H_{1\ 2}$) produce strong and irregular oscillations, trying to pass through almost every point. This behavior indicates overfitting: the model captures the noise rather than the underlying pattern.

The most appropriate hypothesis space seems to be **H_4 or H_5** , as these models follow the general shape of the data without excessive fluctuations. They provide a good balance between simplicity and flexibility, avoiding both underfitting and overfitting.

For this reason, I consider **H_4 or H_5** the most suitable hypothesis space.

C

```
test_data <- read.csv("regr_test.csv")

gg_test <- ggplot(test_data, aes(x, y)) +
  geom_point(color = "black", size = 2) +
  ggtitle("Test data and fitted models (trained on training data)")

x_grid <- seq(min(test_data$x), max(test_data$x), length.out = 200)
plot_data <- data.frame(x = x_grid)

for (i in 1:12) {
  pred <- predict(models[[i]], newdata = plot_data)
  gg_test <- gg_test + geom_line(
```

```
    data = data.frame(x = x_grid, y = pred),
    aes(x = x, y = y),
    color = rainbow(12)[i]
  )
}

print(gg_test)
```

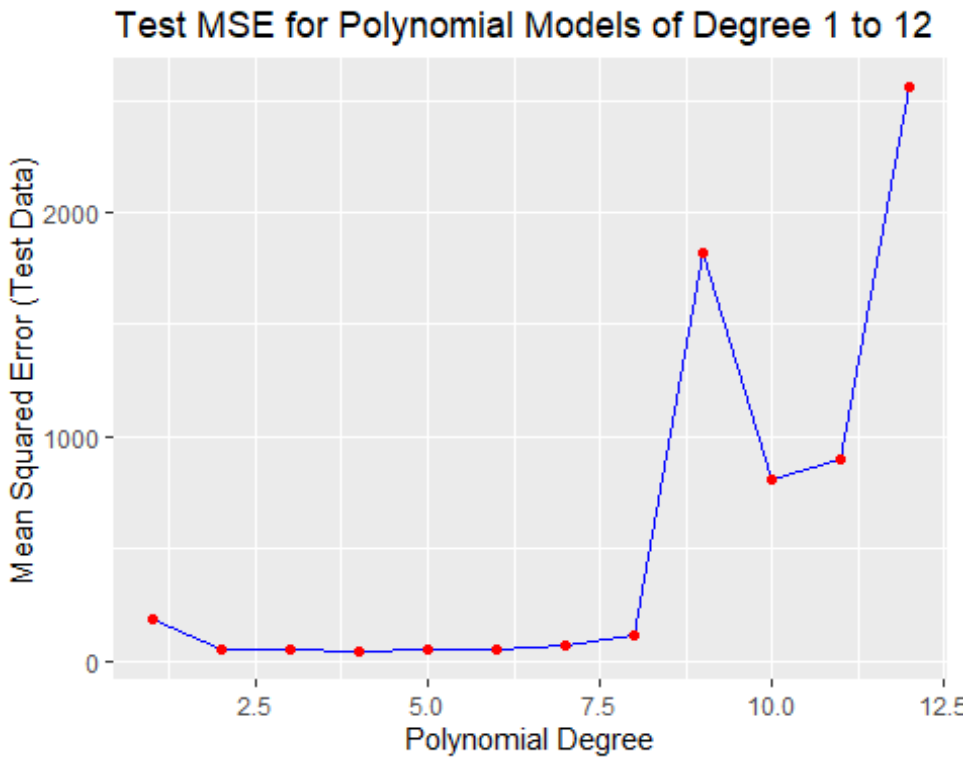
D

```
test_errors <- numeric(12)

for (i in 1:12) {
  predictions <- predict(models[[i]], newdata = test_data)
  test_errors[i] <- mean((test_data$y - predictions)^2)
}

print(test_errors)
```

```
 [1]  183.64140   47.95106   50.86301   40.53414   49.90881   45.90556
 [7]   72.12150  111.78000 1816.93027  810.44548  899.45256 2559.82440
```

```
error_df <- data.frame(
  Degree = 1:12,
  Test_MSE = test_errors
)

ggplot(error_df, aes(x = Degree, y = Test_MSE)) +
```

```
  geom_line(color = "blue") +
  geom_point(color = "red") +
  ggtitle("Test MSE for Polynomial Models of Degree 1 to 12") +
  xlab("Polynomial Degree") +
  ylab("Mean Squared Error (Test Data)")
```



The plot of the test mean squared error (MSE) for polynomial models of degree 1 to 12 shows that the test error decreases initially and remains very low up to around degree 6. However, starting from degree 7, the error begins to rise, and after degree 8, it increases sharply.

This indicates that models with low degrees (1–2) underfit the data and fail to capture the trend, while models of very high degrees (especially 9–12) overfit the training data and perform poorly on the test set. The sudden increase in test error is a typical symptom of overfitting: the model learns the noise in the training data instead of the true underlying relationship.

The best performance is observed between degrees 3 and 6, where the models achieve low test error and generalize well. Therefore, a model of degree 4, 5 or 6 seems to be the most appropriate choice, offering a good balance between underfitting and overfitting.

2

A

```
library(class)

Warning: il pacchetto 'class' è stato creato con R versione 4.4.3
```

Exercise 2: Overfitting and Underfitting: Classification

We want to investigate overfitting and underfitting in a classification example. We will use the k-nearest neighbour classification method. Consider the file "*dat_class.csv*" which contains 2000 observations of two covariates x1 and x2 and a binary response y.

a) **Apply the k-nearest neighbours' method to all the data for k ∈ {1, 5, 10, 20, 50, 100, 200}. Then, for every k, provide two plots side by side. The left plot should always depict the true class for all observations; the right plot should depict the predicted class. Comment on your results.**

With the code from Figure 13 using the KNeighborsClassifier from the sklearn library of Python.

```
from sklearn.neighbors import KNeighborsClassifier

# Load data
data = pd.read_csv("dat_class.csv")
X = data[['x1', 'x2']].values
y = data['y'].values

k_values = [1, 5, 10, 20, 50, 100, 200]

# Plotting function
def plot_true_vs_pred(k):
    knn = KNeighborsClassifier(n_neighbors=k)
    knn.fit(X, y)
    y_pred = knn.predict(X)

    plt.figure(figsize=(8, 4))

    # True class
    plt.subplot(1, 2, 1)
    plt.scatter(X[:, 0], X[:, 1], c=y, cmap='coolwarm', s=5)
    plt.title(f"True Labels (k = {k})")

    # Predicted class
    plt.subplot(1, 2, 2)
    plt.scatter(X[:, 0], X[:, 1], c=y_pred, cmap='coolwarm', s=5)
    plt.title(f"Predicted Labels (k = {k})")

    plt.tight_layout()
    plt.show()

# Loop through k-values and plot
for k in k_values:
    plot_true_vs_pred(k)
```

Figure 13: k-neighbour classifier code

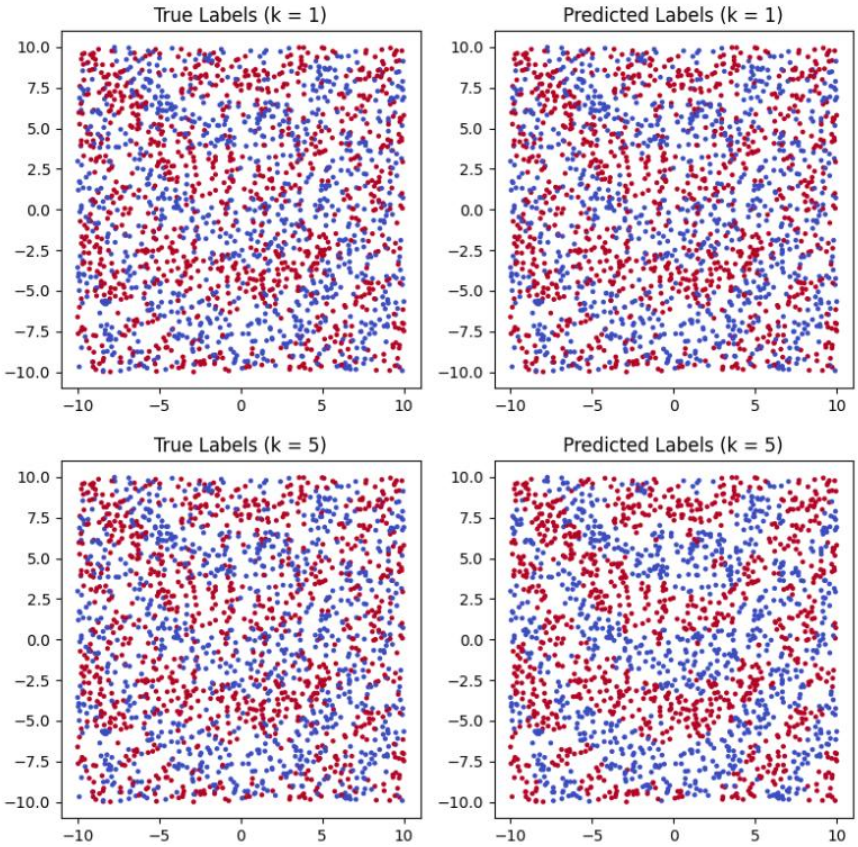


Figure 14: True vs Predicted Labels (k = 1, 5)

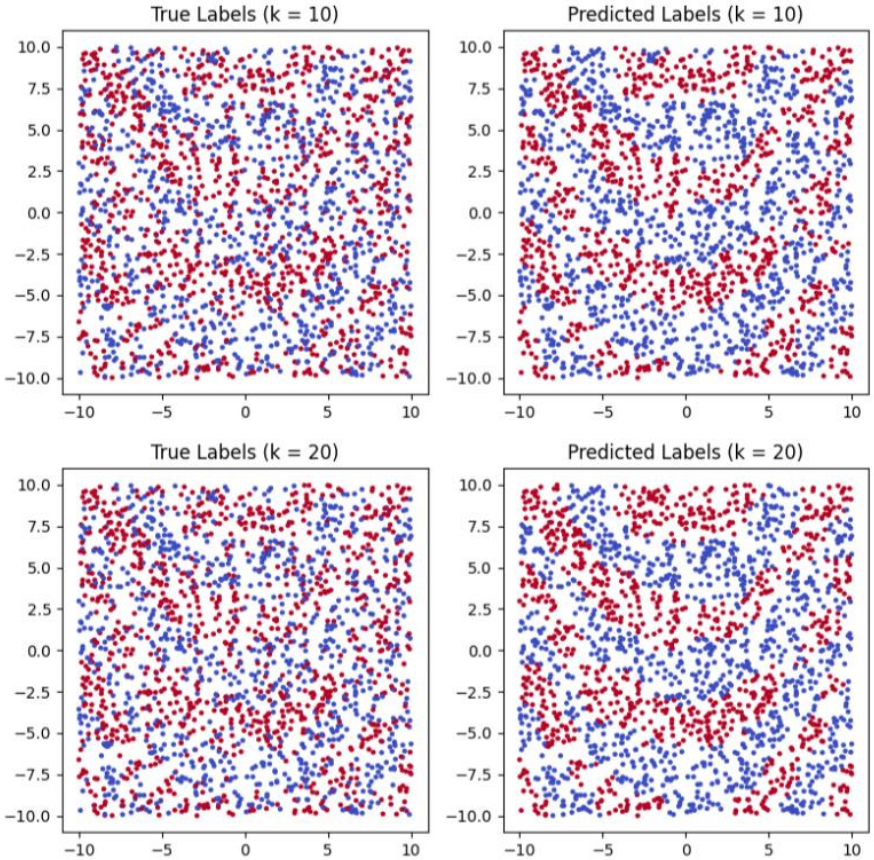


Figure 15: True vs Predicted Labels (k = 10, 20)

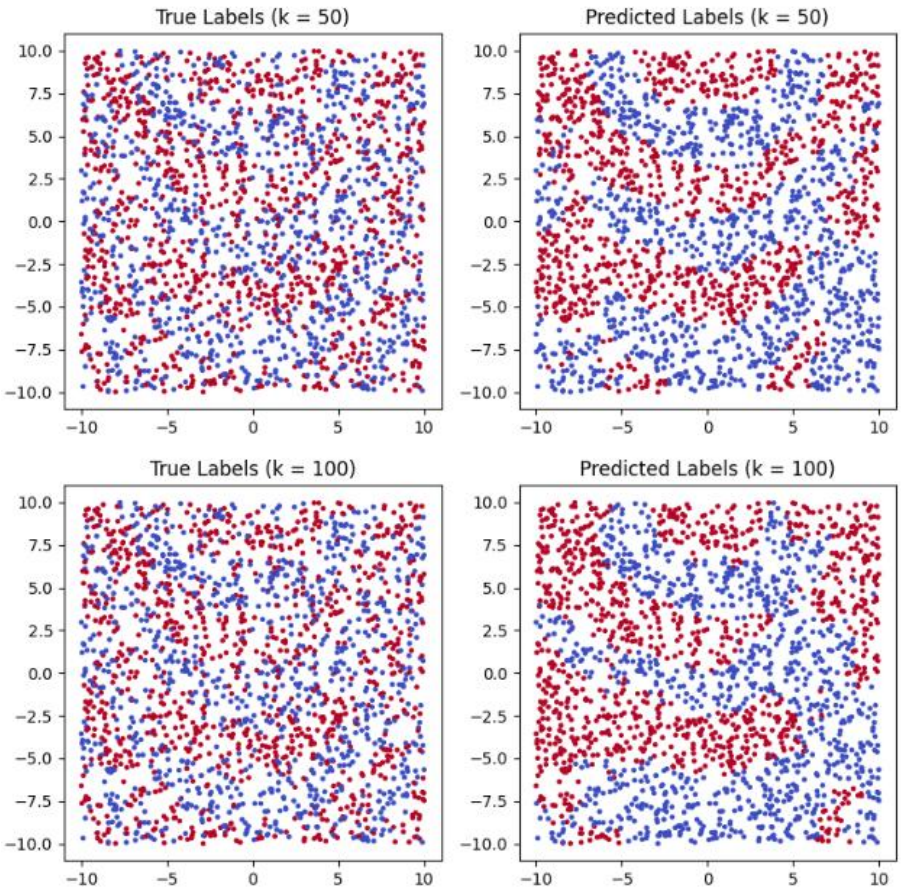


Figure 16: True vs Predicted Labels (k = 50, 100)

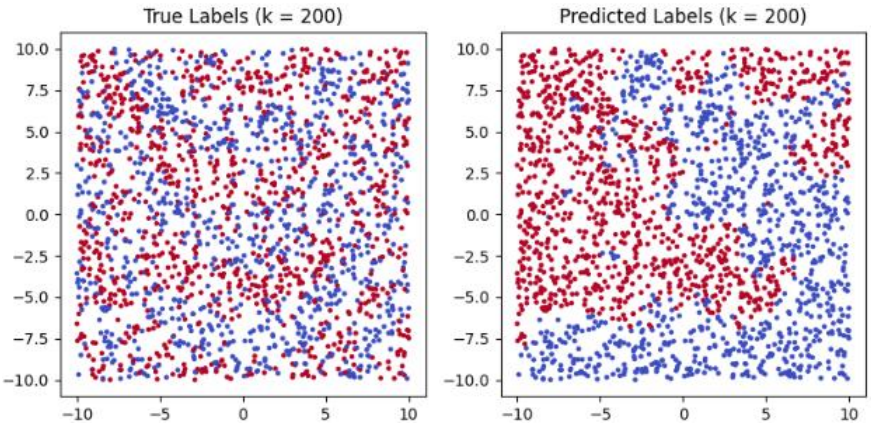


Figure 17: True vs Predicted Labels (k = 200)

The side-by-side plots of true vs predicted class labels for increasing values of $k \in \{1, 5, 10, 20, 50, 100, 200\}$ reveal how the k-nearest neighbours method behaves with respect to model complexity:

- Small k values like the ones from Figure 14**:**
    - The predicted labels closely follow the training data. As a result, we have a low training error, but this can lead to overfitting.
    - The prediction is highly sensitive to local noise, leading to irregular and highly fragmented decision boundaries.

- Moderate k values like the ones from Figure 15 (k not too low but not too high):
  - The predictions begin to smooth out. The model still adapts well to the underlying pattern in the data while ignoring local noise.
  - Therefore, this range appears to have a good balance between bias and variance.
- Large k values from Figure 16 and Figure 17:
  - The predicted labels are increasingly smoothed, and the model becomes more conservative. It starts predicting the majority class more frequently, especially in boundary areas.
  - This will lead to underfitting the data, as the model is ignoring some structures in the data.

b) For k ∈ {1, 2, 3, ..., 50}, estimate the error rates of the model.

First, calculate the training error for each k, i.e., the proportion of observations that get classified wrongly by a k-nearest neighbour model built on the whole data set.

Second, calculate the test error by using subsampling: Randomly split the data into 80% training data and 20% test data, train the model on the training data, test it on the test data. Repeat 20 times and use the mean value over the repetitions as an estimate for the error.

Visualize both error rates as a function of k. What do you observe? How can you explain that?

Running the following Python code from Figure 18

```python
from sklearn.model_selection import train_test_split
from sklearn.metrics import zero_one_loss

ks = range(1, 51)
train_errors = []
test_errors = []

# Repeat 20 times for averaging
n_repeats = 20
rng = np.random.RandomState(0)

for k in ks:
    train_err_k = []
    test_err_k = []

    for _ in range(n_repeats):
        X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=rng)

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

        # Train error on full training set
        y_train_pred = knn.predict(X_train)
        y_test_pred = knn.predict(X_test)

        train_err_k.append(zero_one_loss(y_train, y_train_pred))
        test_err_k.append(zero_one_loss(y_test, y_test_pred))

    train_errors.append(np.mean(train_err_k))
    test_errors.append(np.mean(test_err_k))

# Plot errors
plt.figure(figsize=(8, 5))
plt.plot(ks, train_errors, label='Training error')
plt.plot(ks, test_errors, label='Test error')
plt.xlabel('k (Number of neighbors)')
plt.ylabel('Classification Error')
plt.title('Training and Test Error vs k')
plt.legend()
plt.grid(True)
plt.show()
```

Figure 18: Training and Test Error vs k Python code

We get the plot that can be seen in Figure 19.

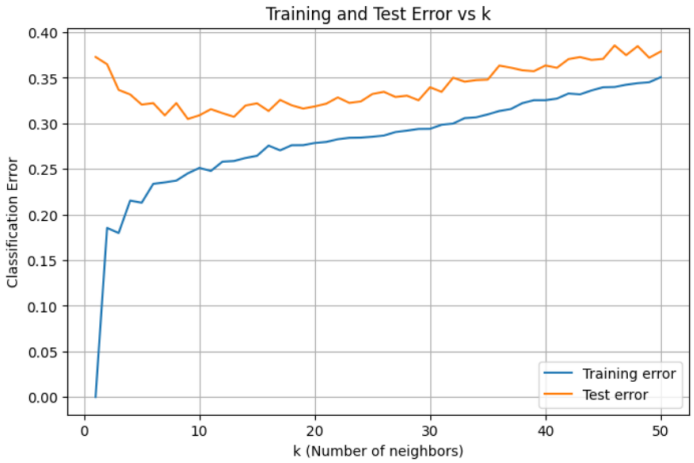


Figure 19: Training and Test Error vs k Python Plot

The Figure 19 shows the classification error for both training and test data as a function of the number of neighbours k in the k-NN model.

On the one hand, we see that the training error **increases with k**.

- For k=1, the training error is very low (≈ 0) because the model perfectly memorizes the training set (each point is its own nearest neighbour).
- As k increases, the model becomes less flexible, and the training error increases steadily.

On the other hand, Test Error:

- The test error initially **decreases** as k increases from 1, reaching a minimum around k = 10/15.
- Beyond this point, the test error **increases again**, which indicates **underfitting**. The model becomes too smooth and fails to capture the structure of the data.

The explanation to this is that:

- Small k: there's a high variance; this is a low bias → overfitting. The model memorizes outliers.
- Large k: the variance gets lower; this is a high bias → underfitting. Too simplistic model.
- And an intermediate value of k achieves the best generalization: balancing the bias and the variance effectively.

To conclude, we can say that the optimal k is around **10 to 15**, where test error is minimized and model generalizes correctly.

c) Decide on an appropriate k and justify your choice. Using this k, apply k-nearest neighbours again to the whole data set and visualize the resulting decision boundaries, i. e., assign all points in R2 (limited to a reasonable range) to one of the two classes.

Based on the test error plot from part b) (Figure 19), we are taking as the optimal choice for k, **k=10**, since it achieves the lowest test error (minimum in the graph) while maintaining a reasonable training error. We think that taking k = 10, the k-NN model generalizes well without being too sensitive to noise in the training data.

So, we apply k-NN with k=10 to the entire dataset, using the code from Figure 20 and visualize the decision boundaries in Figure 21 over a grid of points in $\mathbb{R}^2$.

```python
# Fit model with best k
best_k = 10
knn = KNeighborsClassifier(n_neighbors=best_k)
knn.fit(X, y)

# Grid for visualization
x1_min, x1_max = X[:, 0].min() - 1, X[:, 0].max() + 1
x2_min, x2_max = X[:, 1].min() - 1, X[:, 1].max() + 1
xx, yy = np.meshgrid(np.linspace(x1_min, x1_max, 500),
                     np.linspace(x2_min, x2_max, 500))
grid = np.c_[xx.ravel(), yy.ravel()]
Z = knn.predict(grid).reshape(xx.shape)

# Plot decision boundary
plt.figure(figsize=(6, 5))
cmap_bg = ListedColormap(['#FFAAAA', '#AAAAFF'])
cmap_pts = ListedColormap(['red', 'blue'])

plt.contourf(xx, yy, Z, cmap=cmap_bg, alpha=0.4)
plt.scatter(X[:, 0], X[:, 1], c=y, cmap=cmap_pts, s=5, edgecolor='k')
plt.title(f"Decision Boundary (k = {best_k})")
plt.xlabel("x1")
plt.ylabel("x2")
plt.grid(True)
plt.show()
```

Figure 20: k = 10 decision boundaries visualization Python code

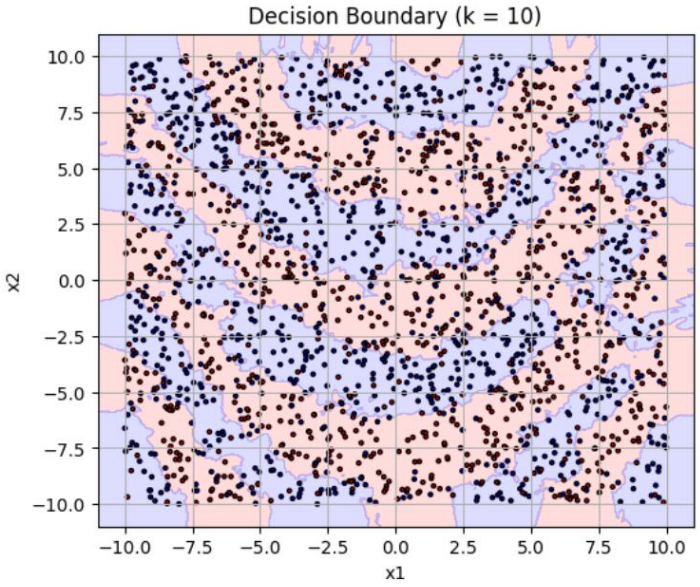
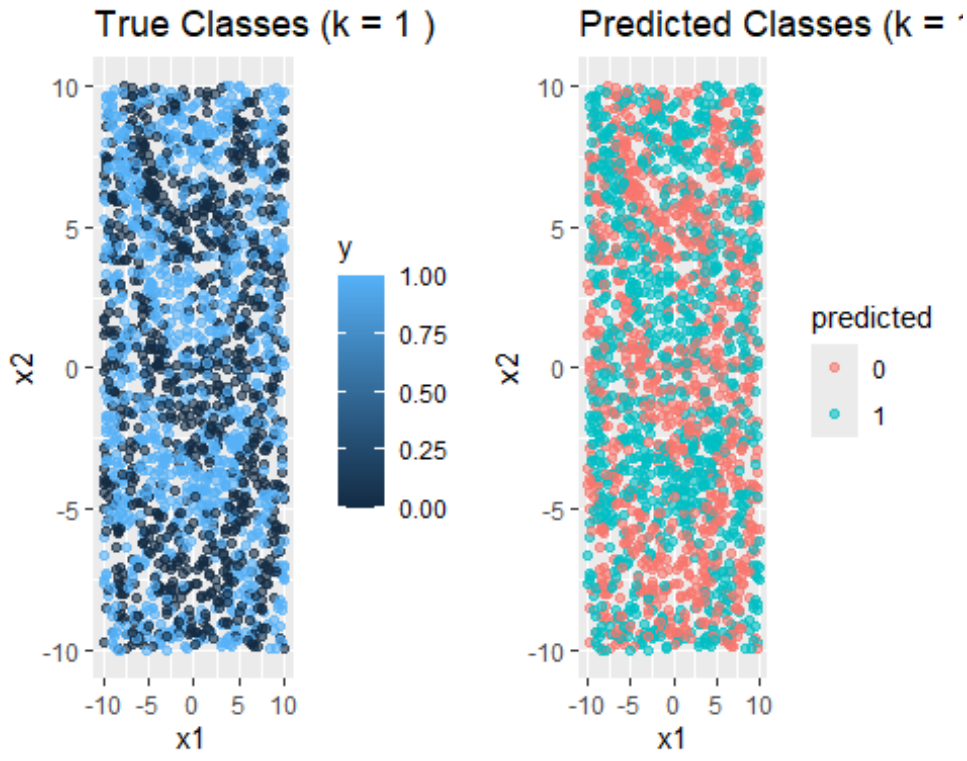
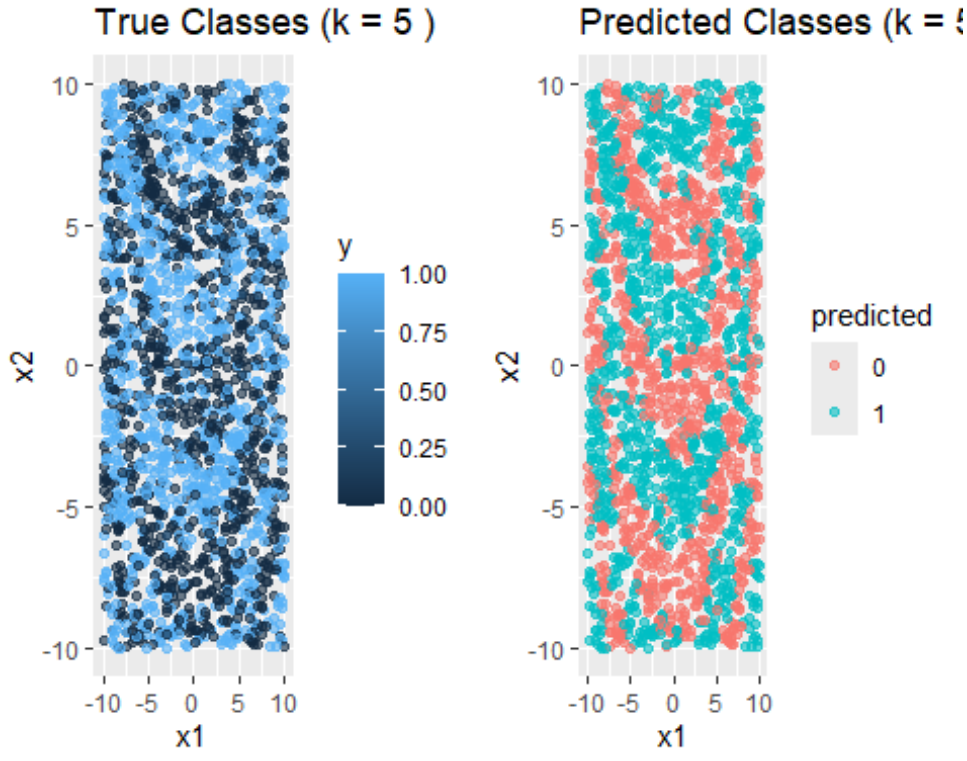


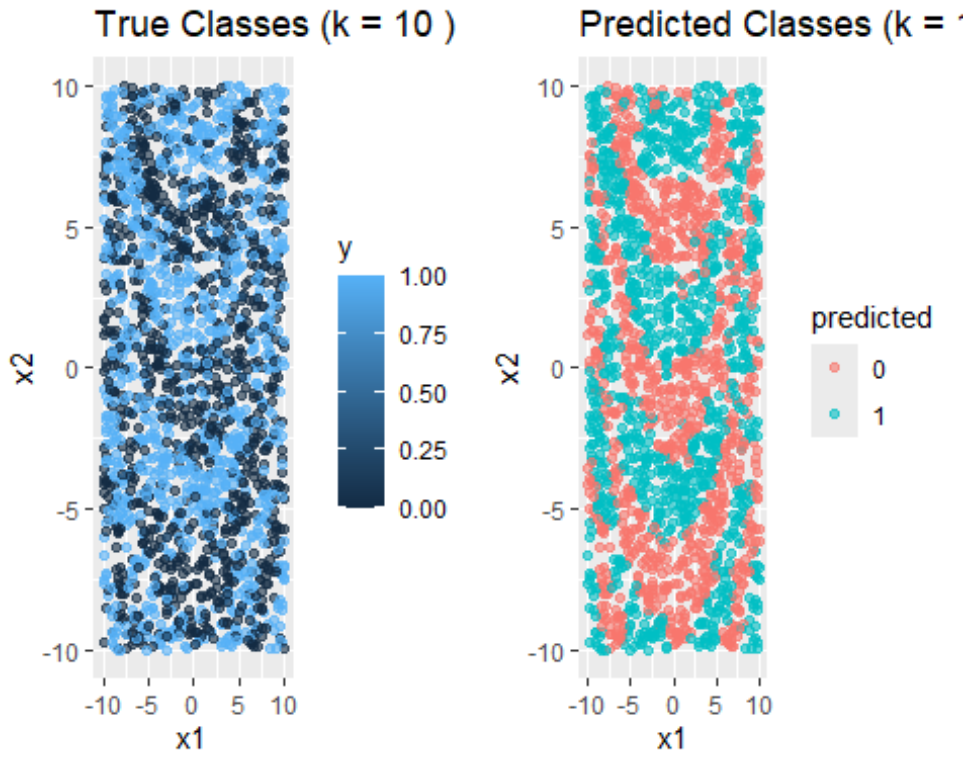
Figure 21: Decision Boundaries Python Plot (k = 10)

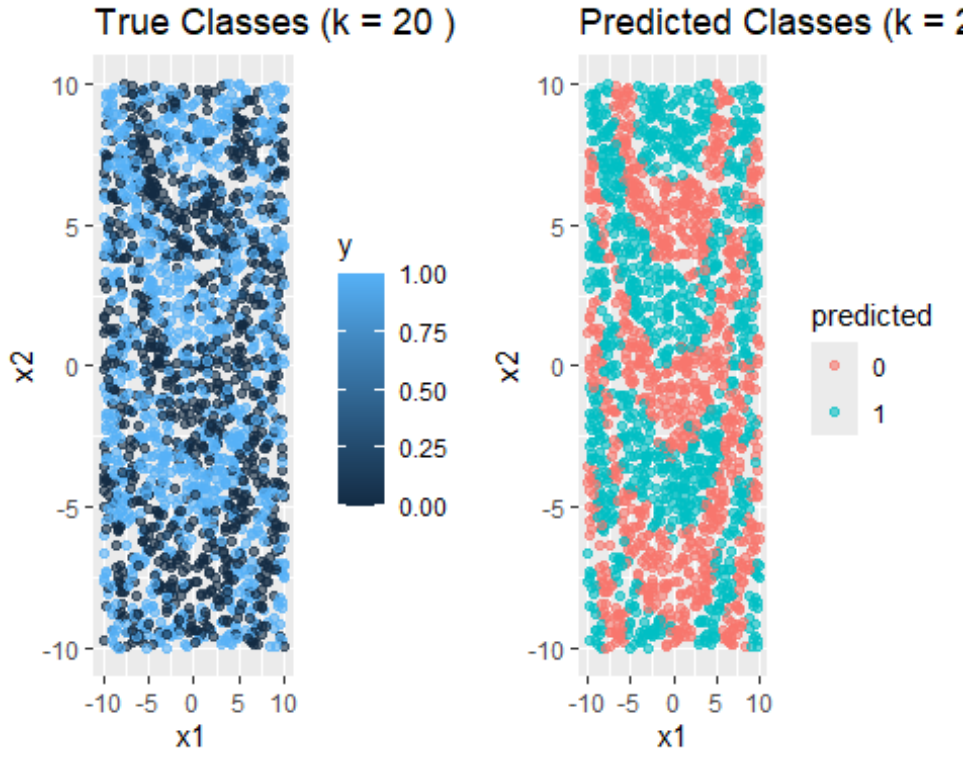
```
library(dplyr)


Caricamento pacchetto: 'dplyr'

I seguenti oggetti sono mascherati da 'package:stats':

    filter, lag

I seguenti oggetti sono mascherati da 'package:base':

    intersect, setdiff, setequal, union
```

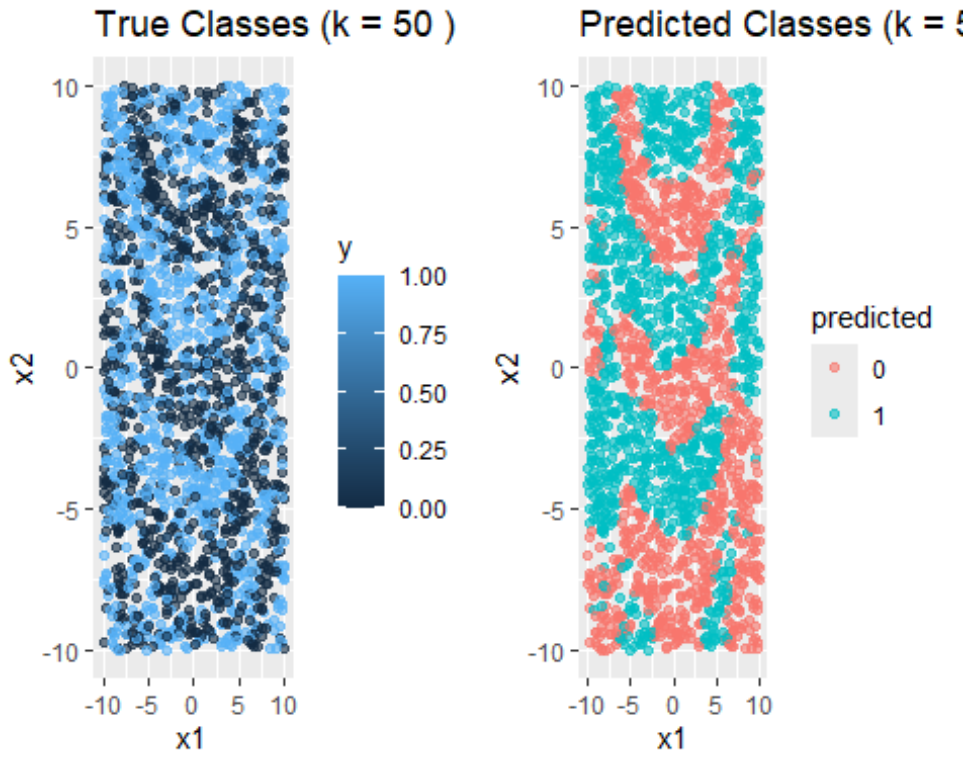
```
data <- read.csv("dat_class.csv")

k_values <- c(1, 5, 10, 20, 50, 100, 200)

x <- data[, c("x1", "x2")]
y <- as.factor(data$y)

plot_knn_result <- function(k) {
  set.seed(123)

  pred <- knn(train = x, test = x, cl = y, k = k)

  data$predicted <- pred

  p_true <- ggplot(data, aes(x = x1, y = x2, color = y)) +
    geom_point(alpha = 0.6) +
    ggtitle(paste("True Classes (k =", k, ")"))

  p_pred <- ggplot(data, aes(x = x1, y = x2, color = predicted)) +
    geom_point(alpha = 0.6) +
    ggtitle(paste("Predicted Classes (k =", k, ")"))

  gridExtra::grid.arrange(p_true, p_pred, ncol = 2)
}

for (k in k_values) {
  plot_knn_result(k)
}
```

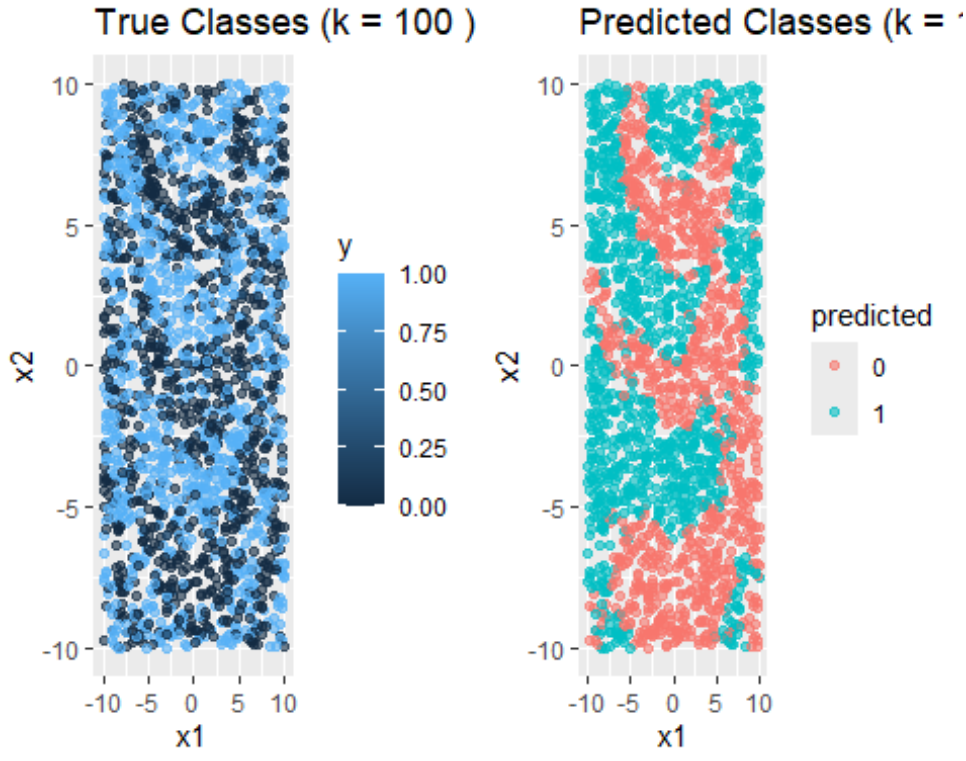

True Classes (k = 1)
Predicted Classes (k =
x2
x1
y
1.00
0.75
0.50
0.25
0.00
predicted
0
1
10
5
0
-5
-10

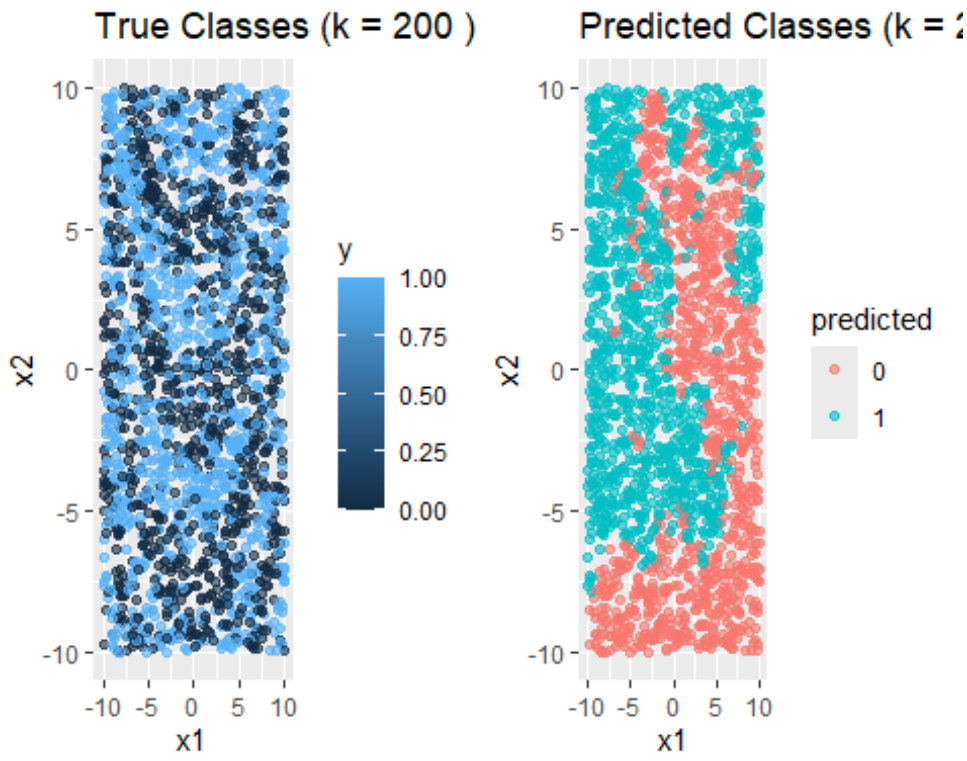

True Classes (k = 5)
Predicted Classes (k = 5
x2
x1
y
1.00
0.75
0.50
0.25
0.00
predicted
0
1
10
5
0
-5
-10


True Classes (k = 10)
Predicted Classes (k =
x2
x1
y
1.00
0.75
0.50
0.25
0.00
predicted
0
1
10
5
0
-5
-10


True Classes (k = 20)
Predicted Classes (k = 2
x2
x1
y
1.00
0.75
0.50
0.25
0.00
predicted
0
1
10
5
0
-5
-10


True Classes (k = 50)
Predicted Classes (k = 5
x2
x1
y
1.00
0.75
0.50
0.25
0.00
predicted
0
1
10
5
0
-5
-10


True Classes (k = 100)
Predicted Classes (k =
x2
x1
y
1.00
0.75
0.50
0.25
0.00
predicted
0
1
10
5
0
-5
-10



As we increase the value of k in the k-nearest neighbors classifier, we can observe clear differences in the predicted class patterns.

For k = 1, the model clearly overfits: it perfectly memorizes the training data and shows a very noisy and irregular class boundary. Each point is classified based on its single nearest neighbor, which causes high sensitivity to noise and local fluctuations.

As k increases (e.g., k = 5, 10, 20), the classification boundaries become smoother and more stable. The model starts to generalize better, balancing flexibility with stability. At these values, the predicted classes start resembling the overall true class distribution more accurately.

With high values of k (e.g., k = 100, 200), the model underfits: it becomes too simplistic and tends to assign the same class to large areas, even when the true data structure is more complex. This results in smooth but inaccurate predictions, especially near the class boundaries.

In conclusion, small k values cause overfitting, large k values cause underfitting, and intermediate values like k = 10 or k = 20 provide the best balance between flexibility and generalization.

B

```
x <- data[, c("x1", "x2")]
y <- as.factor(data$y)

k_values <- 1:50
```

```r
training_errors <- numeric(length(k_values))
test_errors <- numeric(length(k_values))

set.seed(123)

# TRAINING ERROR
for (i in k_values) {
  pred <- knn(train = x, test = x, cl = y, k = i)
  training_errors[i] <- mean(pred != y)
}

# TEST ERROR
repeats <- 20
test_error_matrix <- matrix(0, nrow = repeats, ncol = length(k_values))

for (r in 1:repeats) {
  idx <- sample(1:nrow(data), size = 0.8 * nrow(data))
  x_train <- x[idx, ]
  y_train <- y[idx]
  x_test <- x[-idx, ]
  y_test <- y[-idx]

  for (i in k_values) {
    pred <- knn(train = x_train, test = x_test, cl = y_train, k = i)
    test_error_matrix[r, i] <- mean(pred != y_test)
  }
}

test_errors <- colMeans(test_error_matrix)

library(ggplot2)
error_df <- data.frame(
  k = k_values,
  Training = training_errors,
  Test = test_errors
)

ggplot(error_df, aes(x = k)) +
  geom_line(aes(y = Training, color = "Training Error")) +
  geom_line(aes(y = Test, color = "Test Error")) +
  labs(
    title = "Training and Test Error vs. k",
    x = "k (number of neighbors)",
    y = "Error Rate"
  ) +
  scale_color_manual(values = c("Training Error" = "blue", "Test Error" = "red"))
+
  theme_minimal()
```

The plot shows the training and test error rates as a function of k in the k-nearest neighbors classifier.

For very small k values (especially k = 1), the training error is almost zero, indicating perfect fitting of the training data. However, the test error is relatively high, showing that the model overfits and does not generalize well to new data.

As k increases, the training error steadily rises: the model becomes less flexible and starts to underfit the training data. Meanwhile, the test error initially decreases, reaching a minimum around k ≈ 5–10, and then slowly increases again as k becomes too large.

This behavior illustrates the classic bias-variance tradeoff:
- Small k → low bias but high variance → overfitting.
- Large k → high bias but low variance → underfitting.

The best generalization performance is achieved for intermediate values of k, where the test error is minimized and training and test error are reasonably close.

Therefore, the optimal value of k is typically in the range 5–10, where the model achieves the lowest test error and generalizes well to unseen data.

C

Based on the error plots and the visual inspection of the predicted classes, an appropriate choice for k is **k = 10**. This value achieves a good tradeoff between overfitting and underfitting: the test error is close to its minimum, and the model generalizes well without being too sensitive to noise.

Using k = 10, we apply the k-nearest neighbors method to the entire dataset and visualize the decision boundaries by predicting the class of every point in a grid over the input space. This helps us understand how the model would classify new data in different regions of the input space.

```
k_chosen <- 10

x1_range <- seq(-10, 10, length.out = 200)
x2_range <- seq(-10, 10, length.out = 200)
grid <- expand.grid(x1 = x1_range, x2 = x2_range)

knn_preds <- knn(train = data[, c("x1", "x2")],
                 test = grid,
                 cl = as.factor(data$y),
                 k = k_chosen)

grid$predicted <- knn_preds

ggplot() +
  geom_tile(data = grid, aes(x = x1, y = x2, fill = predicted), alpha = 0.4) +
  geom_point(data = data, aes(x = x1, y = x2, color = as.factor(y)), size = 1.5) +
  scale_fill_manual(values = c("0" = "salmon", "1" = "lightblue")) +
  scale_color_manual(values = c("0" = "red", "1" = "blue")) +
  labs(title = paste("Decision Boundaries (k =", k_chosen, ")"),
       fill = "Predicted", color = "True") +
  theme_minimal()
```