Intro to ML: Homework 0

Carlos Gonzalez Rivera

September 1, 2023

1 Programming KNN

Results of both KNN model types using both implementations (built-in and from-scratch) with both simulated and real-world datasets as k values change are attached at the end as the cells and output to the Jupyter Notebook.

2 Linear Regression

2.1 Ordinary Least Squares Problem

The regression model is given by $y = X\beta + e$, where $y \in \mathbb{R}^n$, $X \in \mathbb{R}^{n \times p}$ has rank $p < n$, $\beta \in \mathbb{R}^p$, and $e \in \mathbb{R}^n$ is a random vector. The ordinary least squares (OLS) estimator is defined as the argument that minimizes the sum of squared residuals:

$$\hat{\beta}_{\text{OLS}} = \arg\min_{\beta} \|y - X\beta\|_2^2 \tag{1}$$

The assumptions for the random vector e are $E(e) = 0$ and $\text{cov}(e) = \sigma^2 I$.

2.2 Expression for $\hat{\beta}_{\text{OLS}}$

To find the expression for $\hat{\beta}_{\text{OLS}}$, we start with the objective function to minimize the sum of squared residuals:

$$\|y - X\beta\|_2^2 = (y - X\beta)^T(y - X\beta) \tag{2}$$

$$= y^T y - y^T X\beta - \beta^T X^T y + \beta^T X^T X\beta \tag{3}$$

We take the derivative with respect to β:

$$\frac{d}{d\beta}\left(y^T y - y^T X\beta - \beta^T X^T y + \beta^T X^T X\beta\right) = -2X^T y + 2X^T X\beta \tag{4}$$

Setting the derivative equal to zero and solving for β, we get:

$$2X^T y = 2X^T X\beta \tag{5}$$

$$X^T y = X^T X\beta => \beta = (X^T X)^{-1} X^T y \tag{6}$$

$$\therefore \hat{\beta}_{\text{OLS}} = (X^T X)^{-1} X^T y \tag{7}$$

2.3 Expected Value $E(\hat{\beta}_{\text{OLS}})$

The expected value of $\hat{\beta}_{\text{OLS}}$ is calculated as follows:

$$E(\hat{\beta}_{\text{OLS}}) = E((X^T X)^{-1} X^T y) \tag{8}$$

$$= E((X^T X)^{-1} X^T (X\beta + e)) \tag{9}$$

$$= E((X^T X)^{-1} X^T X\beta + (X^T X)^{-1} X^T e) \tag{10}$$

$$= (X^T X)^{-1} X^T X\beta + (X^T X)^{-1} X^T E(e) \tag{11}$$

$$= \frac{\beta(X^T X)}{(X^T X)} + 0, \text{ given that E(e)} = 0 \tag{12}$$

$$= \beta \tag{13}$$

Given $E(e) = 0$, the expectation of the OLS estimator is equal to the true parameter value β, confirming that the estimator is unbiased.

2.4 Covariance $\text{cov}(\hat{\beta}_{\text{OLS}})$

The covariance of $\hat{\beta}_{\text{OLS}}$ is calculated as:

$$\text{cov}(\hat{\beta}_{\text{OLS}}) = \text{cov}((X^T X)^{-1} X^T (X\beta + e)) \tag{14}$$

$$= \text{cov}((X^T X)^{-1} X\beta + (X^T X)^{-1} X^T e) \tag{15}$$

$$= (X^T X)^{-1} X^T X (X^T X)^{-1} \text{cov}(e) \tag{16}$$

$$= \frac{(X^T X)}{(X^T X)} \sigma^2 I \tag{17}$$

$$= \sigma^2 (X^T X)^{-1} \tag{18}$$

Given that $\text{cov}(e) = \sigma^2 I$, the covariance of the OLS estimator is $(X^T X)^{-1}\sigma^2$. This provides an expression for its variability as $I = (X^T X)^{-1}$.

2.5 OLS Predictions $\hat{y} = Hy$

The OLS predictions $\hat{y} = X\hat{\beta}_{\text{OLS}}$ can be expressed in the form $\hat{y} = Hy$ for some matrix H. Specifically, $H = X(X^TX)^{-1}X^T$ is often referred to as the "hat matrix" because it "puts the hat" on the vector y, transforming it into the predicted values $\hat{y}$. Given the OLS estimator $\hat{\beta}_{\text{OLS}} = (X^TX)^{-1}X^Ty$, we can write the predictions as:

$$\begin{aligned} \hat{y} &= X\hat{\beta}_{\text{OLS}} && (19)\\ &= X(X^TX)^{-1}X^Ty && (20)\\ &= Hy && (21) \end{aligned}$$

3 Matrix Analysis

3.1 Mean Transformation

We want $E(\tilde{x}) = 0$, so we'll need to find A and b such that:

$$E(\tilde{x}) = E(Ax + b) = AE(x) + b = A\mu + b = 0 \quad (22)$$

Given the known mean $E(x) = \mu$, we can solve for b:

$$b = -A\mu \quad (23)$$

3.2 Covariance Transformation

Next, we want $\text{cov}(\tilde{x}) = I$, so:

$$\text{cov}(\tilde{x}) = \text{cov}(Ax + b) = \text{cov}(Ax) = I \quad (24)$$

Recalling $\text{cov}(x) = E[(x - \mu)(x - \mu)^T]$, we must find $\text{cov}(y)$ with:

$$cov(y) = E[(y - E[y])(y - E[y])^T] \quad (25)$$

Since $y = Ax$ and $E[y] = AE[x] = A\mu$,

$$\begin{aligned} cov(y) &= E[(Ax - A\mu)(Ax - A\mu)^T] && (26)\\ &= E[A(x - \mu)(A^T)(x - \mu)^T] && (27)\\ &= AE[(x - \mu)(x - \mu)^T]A^T && (28)\\ &= A\Sigma A^T, \text{ given } \Sigma = E[(x - \mu)(x - \mu)^T] && (29)\\ & && (30) \end{aligned}$$

Q is an orthogonal matrix whose columns are the eigenvectors of Σ and Λ is a diagonal matrix with eigenvectors $\lambda_1, \lambda_2, \lambda_3, ..., \lambda_p$ on its diagonal. A positive definite matrix Σ can be eigen-decomposed as:

$$Q\Lambda Q^T = \Sigma \tag{31}$$

$$\therefore Q\Lambda^{-\frac{1}{2}} Q^T = \Sigma^{-\frac{1}{2}} \tag{32}$$

Proving that $A\Sigma A^T = I$ when $A = \Sigma^{-\frac{1}{2}}$:

$$\Sigma^{-\frac{1}{2}}\Sigma\Sigma^{-\frac{1}{2}} = (Q\Lambda^{-\frac{1}{2}}Q^T)(Q\Lambda Q^T)(Q\Lambda^{-\frac{1}{2}}Q^T) \tag{33}$$

$$= Q\Lambda^{-\frac{1}{2}}Q^TQ\Lambda Q^TQ\Lambda^{-\frac{1}{2}}Q^T \tag{34}$$

$$\tag{35}$$

Since the identity matrix I acts as the multiplicative identity in matrix multiplication, meaning $AI = IA = A$ for any matrix A, we know $Q^TQ = I$ because Q an orthogonal matrix:

$$= Q\Lambda^{-\frac{1}{2}}I\Lambda I\Lambda^{-\frac{1}{2}}Q^T \tag{36}$$

$$= Q\Lambda^{-\frac{1}{2}}\Lambda\Lambda^{-\frac{1}{2}}Q^T \tag{37}$$

$$= QIQ^T \tag{38}$$

$$= II = I \tag{39}$$

Given Σ is positive definite, we can solve for A:

$$A = \Sigma^{-\frac{1}{2}} \tag{40}$$

3.3 Final Transformation

Putting it all together, we find the affine transformation:

$$A = \Sigma^{-\frac{1}{2}} \tag{41}$$

$$b = -A\mu = -\Sigma^{-\frac{1}{2}}\mu \tag{42}$$

Thus, the desired transformation is:

$$\tilde{x} = \Sigma^{-\frac{1}{2}}x - \Sigma^{-\frac{1}{2}}\mu, \text{ from } \tilde{x} = Ax + b \tag{43}$$

$$\tilde{x} = \Sigma^{-\frac{1}{2}}(x - \mu) \tag{44}$$

3.4 Outer Product of xy^T

Letting $x, y \in \mathbb{R}^n$, the goal is to find the eigenvalues and eigenvectors of the outer product xy^T. The outer product xy^T results in an $n \times n$ matrix, denoted as M:

$$M = xy^T \tag{45}$$

3.5 Eigenvalues and Eigenvectors of xy^T

Since the rank of M is at most 1, at most one eigenvalue will be non-zero, and the rest will be zero. The non-zero eigenvalue is given by the inner product of the vectors x and y, i.e., $\lambda = x \cdot y$. The eigenvector corresponding to the non-zero eigenvalue λ is the vector x itself.

$$\text{Eigenvalues: } \lambda = x \cdot y \tag{46}$$

$$\text{Eigenvectors: } x \tag{47}$$

3.6 Sum of Squares of Elements and Eigenvalues of a Real Symmetric Matrix

Let A be a real symmetric $p \times p$ matrix with eigenvalues $\lambda_1, \ldots, \lambda_p$. The goal is to prove that $\sum_{i=1}^{p} \sum_{j=1}^{p} a_{ij}^2 = \sum_{i=1}^{p} \lambda_i^2$.

3.7 Diagonalization of A and Orthogonality of Q

Since A is real and symmetric, it can be diagonalized by an orthogonal matrix Q. The orthogonality of Q implies $Q^T Q = QQ^T = I$.

$$A = Q\Lambda Q^T = I\Lambda \tag{48}$$

3.8 Computing the Frobenius Norm

The Frobenius norm of A is defined as $\|A\|_F = \sqrt{\sum_{i=1}^{p} \sum_{j=1}^{p} a_{ij}^2}$. Using the diagonalization, we find:

$$\begin{aligned}
\|A\|_F^2 &= \|Q\Lambda Q^T\|_F^2 && (49)\\
&= \text{trace}((Q\Lambda Q^T)^T(Q\Lambda Q^T)) && (50)\\
&= \text{trace}(Q\Lambda^T Q^T Q\Lambda Q^T) && (51)\\
&= \text{trace}(Q\Lambda^T \Lambda Q^T) && (52)\\
&= \text{trace}(\Lambda^T \Lambda) && (53)\\
&= \sum_{i=1}^{p} \lambda_i^2 && (54)
\end{aligned}$$

The trace of a square matrix is the sum of its diagonal elements. It's denoted as trace(A) and are defined as $\Sigma_{i=1}^{n} a_{ii}$ with useful properties like $trace(AB) = trace(BA)$ and $trace(A) = trace(A^T)$

3.9 Conclusion

Therefore, the sum of the squares of the elements of A is equal to the sum of the squares of its eigenvalues.

$$||A||_F^2 = \sum_{i=1}^{p} \sum_{j=1}^{p} a_{ij}^2 = \sum_{i=1}^{p} \lambda_i^2 \quad (55)$$

4 Multivariate Statistics

4.1 Joint Likelihood Function

The likelihood function ($L(\Sigma)$) for p-variate normal distribution with mean 0 and covariance Σ is:

$$L(\Sigma|\mathbf{x_1}, \mathbf{x_2}, \ldots, \mathbf{x_n}) = \prod_{i=1}^{n} \frac{1}{(2\pi)^{p/2}|\Sigma|^{1/2}} \exp\left(-\frac{1}{2}\mathbf{x_i}^T\Sigma^{-1}\mathbf{x_i}\right)$$

4.2 Log-Likelihood Function

Taking the logarithm of the joint likelihood function gives the log-likelihood function:

$$\begin{aligned}
\log L(\Sigma) &= \log\left(\prod_{i=1}^{n} \frac{1}{(2\pi)^{p/2}|\Sigma|^{1/2}} \exp\left(-\frac{1}{2}\mathbf{x_i}^T\Sigma^{-1}\mathbf{x_i}\right)\right) \\
&= \log\left(-\frac{p}{2}\log(2\pi) - \frac{1}{2}\log(|\Sigma|) - \frac{1}{2}\sum_{i=1}^{n}\mathbf{x_i}^T\Sigma^{-1}\mathbf{x_i}\right) \\
&= -\frac{np}{2}\log(2\pi) - \frac{n}{2}\log(|\Sigma|) - \frac{1}{2}\sum_{i=1}^{n}\mathbf{x_i}^T\Sigma^{-1}\mathbf{x_i} \\
&= -\frac{n}{2}\log|\Sigma| - \frac{1}{2}\sum_{i=1}^{n}\mathbf{x_i}^T\Sigma^{-1}\mathbf{x_i}
\end{aligned}$$

4.3 Gradient of Log-Likelihood (Score Equations)

To maximize the log-likelihood function, take the gradient with respect to Σ^{-1},also called the precision matrix Θ, ($\Theta = \Sigma^{-1}$) and set it equal to zero:

$$\begin{aligned}
\log L(\Theta) &= -\frac{n}{2}\log|\Theta^{-1}| - \frac{1}{2}\sum_{i=1}^{n} \mathbf{x_i}^T \Theta \mathbf{x_i} \\
\frac{\partial}{\partial\Theta}\log L &= \frac{n}{2}\Theta^{-1} - \frac{1}{2}\sum_{i=1}^{n} \mathbf{x_i}\mathbf{x_i}^T \\
\frac{n}{2}\Theta^{-1} &= \frac{1}{2}\sum_{i=1}^{n} \mathbf{x_i}\mathbf{x_i}^T \\
\Theta &= n\left(\sum_{i=1}^{n} \mathbf{x_i}\mathbf{x_i}^T\right)^{-1}
\end{aligned}$$

4.4 Maximum Likelihood Estimate of Σ

Finally, the MLE of Σ can be found by taking the inverse of $\hat{\Theta}$:

$$\begin{aligned}
\hat{\Sigma} &= \left(\hat{\Theta}\right)^{-1} \\
&= \left(n\left(\sum_{i=1}^{n} \mathbf{x_i}\mathbf{x_i}^T\right)^{-1}\right)^{-1} \\
&= \frac{1}{n}\sum_{i=1}^{n} \mathbf{x_i}\mathbf{x_i}^T \\
\hat{\Sigma} &= \frac{\mathbf{X}^T\mathbf{X}}{n}
\end{aligned}$$

5 Convex Optimization

5.1 (a) $\max(0, 1 - y\alpha^\top x)$ for $y \in \{-1, 1\}$ Analysis

5.1.1 Case $y = 1$

The function becomes $f(x) = \max(0, 1 - \alpha^T x)$.

1. When $1 - \alpha^T x \leq 0$, the function is $f(x) = 0$, which is a constant function and therefore convex (but not strictly convex).

2. When $1 - \alpha^T x > 0$, the function is $f(x) = 1 - \alpha^T x$, which is a linear function and therefore also convex (but not strictly convex).

Since both segments of the piecewise function are convex, the overall function for $y = 1$ is convex.

5.1.2 Case $y = -1$

The function becomes $f(x) = \max(0, 1 + \alpha^T x)$.

1. When $1 + \alpha^T x \leq 0$, the function is $f(x) = 0$, which is a constant function and therefore convex (but not strictly convex).

2. When $1 + \alpha^T x > 0$, the function is $f(x) = 1 + \alpha^T x$, which is a linear function and therefore also convex (but not strictly convex).

Since both segments of the piecewise function are convex, the overall function for $y = -1$ is convex.

5.1.3 Case $y = 0$

the function simplifies to: $f(x) = max(0, 1 - 0\alpha^T x) = max(0, 1)$. In this case, the function becomes a constant function $f(x) = 1$, regardless of the value of x. Constant functions are convex (not concave), but not strictly convex. It is also readily known the second derivative of a constant is zero, which satisfies the condition for convexity: $f''(x) = 0 \geq 0$. So for $y = 0$, the function is also convex but not strictly convex.

5.1.4 Summary

For both $y = 1$ and $y = -1$, the function $f(x) = \max(0, 1 - y\alpha^T x)$ is convex but not strictly convex. It is also not concave. Therefore, the function is convex for $x \in \mathbb{R}^n$ and $y \in \{-1, 1\}$ (although not strictly convex).

5.2 (b) $\log(1 + e^{\alpha^\top x})$ Analysis:

$$f'(x) = \frac{e^{\alpha^\top x}\alpha}{1 + e^{\alpha^\top x}}, \text{ by applying the chain rule}$$
$$f''(x) = \frac{\alpha\alpha^\top e^{\alpha^\top x}}{(1 + e^{\alpha^\top x})^2}, \text{ by applying the quotient rule}$$

The convexity of the function depends on the sign of α. Specifically:

- The function is not strictly convex when α is a zero vector, as $f''(x)$ would be zero.
- The function is convex for $x \in \mathbb{R}^n$ when α is a zero vector or a non-zero vector.

5.3 (c) $-y\log(\alpha^\top x) - (1-y)\log(1-\alpha^\top x)$ for $y \in \{0, 1\}$ Analysis:

5.3.1 Case $y = 0$:

The function becomes $-(1-y)\log(1-\alpha^\top x)$ when $y = 0$. Therefore, the first derivative of this new function for $y = 0$ should be:

$$f'(x) = \frac{\alpha^\top}{\ln(10)(1 - \alpha^\top x)} \tag{56}$$

The second derivative of this function for $y = 0$ is then:

$$f''(x) = -\frac{\alpha^{\top 2}}{\ln(10)(1 - \alpha^\top x)^2} \tag{57}$$

5.3.2 Case $y = 1$:

The function becomes $-y \log(\alpha^\top x)$ when $y = 1$. The first derivative of this new function for $y = 1$ should be:

$$f'(x) = -\frac{1}{x \ln(10)} \tag{58}$$

The second derivative of this function for $y = 1$ is then:

$$f''(x) = \frac{1}{x^2 \ln(10)} \tag{59}$$

5.3.3 Summary

This function is commonly known as the logistic loss. Its second derivative doesn't yield a sign-definite expression (dependent on which y value is used for estimation), and thus the function is either globally convex or concave. For instance, $y = 0$ returns a concave solution while $y = 1$ returns convexity strictness.

5.4 (d) $\|y - Ax\|_1$ Analysis:

The ℓ_1 norm is actually a convex function, but it's not strictly convex because the second derivative doesn't exist everywhere due to the absolute value operation. The function is a sum of absolute value functions, which are convex. The function has a non-empty subdifferential at every point, proving it is convex. However, the function is not strictly convex due to flat regions.

5.5 (e) $\|y - Ax\|_{2,2} + \|x\|_{2,2}$

Analysis: Both the ℓ_2 norms and their sum are convex, and their Hessians are positive definite. Both the Euclidean (ℓ_2) norm and the square of the Euclidean norm are convex. However, the function is not strictly convex because Euclidean norm has points where the second derivative is zero.

$$\theta \in [0, 1], f(\theta x_1 + (1 - \theta)x_2) \leq \theta f(x_1) + (1 - \theta) f(x_2). \tag{60}$$

$$\|\theta u + (1 - \theta)v\|_2 \leq \theta \|u\|_2 + (1 - \theta)\|v\|_2 \tag{61}$$

6 Programming Convex Optimization

Mathematical formulae for the gradients of the functions described for parts (c) and (e). Pseudo-code attached at the end of the Jupyter Notebook (as last two Python cells and corresponding output).

Gradient for Part (c): $-y\log(\alpha^\top x) - (1-y)\log(1-\alpha^\top x)$ for $y \in \{0,1\}$

The gradient of $f(x) = -y\log(\alpha^\top x) - (1-y)\log(1-\alpha^\top x)$ is given by:

$$\nabla f(x) = -y\frac{\alpha}{\alpha^\top x} + (1-y)\frac{\alpha}{1-\alpha^\top x}$$

Gradient for Part (e): $\|y - Ax\|_{2,2} + \|x\|_{2,2}$

The gradient of $f(x) = \|y - Ax\|_{2,2} + \|x\|_{2,2}$ is given by:

$$\nabla f(x) = 2A^\top(Ax - y) + 2x$$

K-NN Implementations from Scratch

Initialization of Utility Functions (Shuffling, Mean, Distance)

In [1]:

```
# Linear Congruential Generator (LCG)
# for pseudo-random numbers
def lcg(seed, a, c, m, n):
    numbers = []
    x = seed
    for _ in range(n):
        x = (a * x + c) % m
        numbers.append(x)
    return numbers
```

In [2]:

```
# Fisher-Yates Shuffle using LCG
def fisher_yates_shuffle(lst, seed=1):
    a, c, m = 1664525, 1013904223, 2**32
    n = len(lst)
    random_numbers = lcg(seed, a, c, m, n)
    for i in range(n - 1, 0, -1):
        j = random_numbers[n - 1 - i] % (i + 1)
        lst[i], lst[j] = lst[j], lst[i]
    return lst
```

In [3]:

```
def euclidean_distance(x1, x2):
    return sum((a - b)**2 for a, b in zip(x1, x2)) ** 0.5
```

In [4]:

```
def most_common(lst):
    return max(set(lst), key=lst.count)
```

In [5]:

```
def mean(lst):
    return sum(lst) / len(lst)
```

Regression K-NN Initialization

In [6]:

```
def knn_regression(X_train, y_train, X_test, k):
    y_pred = []

    for test_point in X_test:
        distances = [euclidean_distance(test_point, train_point) for train_point in X_tr
ain]
        k_indices = sorted(range(len(distances)), key=lambda i: distances[i])[:k]
        k_nearest_values = [y_train[i] for i in k_indices]
        y_pred.append(mean(k_nearest_values))

    return y_pred
```

Create and shuffle the training data for regression

In [7]:

```
X_train_regress = [[i, j] for i in range(-5, 6) for j in range(-5, 6)]
y_train_regress = [2 * i + 3 * j + (i % 2 - 0.5) * 2 for i, j in X_train_regress]

combined_regress = list(zip(X_train_regress, y_train_regress))

shuffled_combined_regress = fisher_yates_shuffle(combined_regress.copy())

X_train_regress_shuffled, y_train_regress_shuffled = zip(*shuffled_combined_regress)
```

Classification K-NN Initialization

In [8]:

```
def knn_classification(X_train, y_train, X_test, k):
    y_pred = []

    for test_point in X_test:
        distances = [euclidean_distance(test_point, train_point) for train_point in X_tr
ain]
        k_indices = sorted(range(len(distances)), key=lambda i: distances[i])[:k]
        k_nearest_labels = [y_train[i] for i in k_indices]
        y_pred.append(most_common(k_nearest_labels))

    return y_pred
```

Create and shuffle the training data for classification

In [9]:

```
X_train_class = [[i, j] for i in range(-5, 6) for j in range(-5, 6)] + [[i, j] for i in
range(10, 16) for j in range(10, 16)]
y_train_class = [0 for _ in range(121)] + [1 for _ in range(36)]

combined_class = list(zip(X_train_class, y_train_class))

shuffled_combined_class = fisher_yates_shuffle(combined_class.copy())

X_train_class_shuffled, y_train_class_shuffled = zip(*shuffled_combined_class)
```

Testing "manual" KNN Implementations

Classification Testing Split

In [10]:

```
X_test_class = [[i, i] for i in range(-50, 50, 1)]

shuffled_X_test_class = fisher_yates_shuffle(X_test_class.copy())

shuffled_y_test_class = [1 if x[0] > 5 and x[1] > 5 else 0 for x in shuffled_X_test_clas
s]
```

Regression Testing Split

In [11]:

```
X_test_regress = [[i, i] for i in range(-50, 50, 1)]

shuffled_X_test_regress = fisher_yates_shuffle(X_test_regress.copy())

shuffled_y_test_regress = [2 * x[0] + 3 * x[1] + (x[0] % 2 - 0.5) * 2 for x in shuffled_
X_test_regress]
```

Apply custom K-NN implementations for k=3 on newly pseudo-shuffled datasets

In [12]:

```
k = 3
```

Regression

In [13]:

```
knn_regression(X_train_regress_shuffled, y_train_regress_shuffled, shuffled_X_test_regres
s, k)
```

Out[13]:

```
[23.666666666666668,
 -23.0,
 23.666666666666668,
 23.666666666666668,
 -23.0,
 23.666666666666668,
 -23.0,
 -23.0,
 23.666666666666668,
 20.0,
 -23.0,
 -23.0,
 -23.0,
 -23.0,
 -15.333333333333334,
 -23.0,
 -23.0,
 23.666666666666668,
 -23.0,
 23.666666666666668,
 -23.0,
 16.0,
 23.666666666666668,
 23.666666666666668,
 23.666666666666668,
 23.666666666666668,
 23.666666666666668,
 23.666666666666668,
 -23.0,
 23.666666666666668,
 23.666666666666668,
 23.666666666666668,
 23.666666666666668,
 -23.0,
 23.666666666666668,
 23.666666666666668,
 -23.0,
 23.666666666666668,
 23.666666666666668,
 23.666666666666668,
 -23.0,
 23.666666666666668,
 23.666666666666668,
 -23.0,
 -23.0,
 23.666666666666668,
 -23.0,
 23.666666666666668,
 -23.0,
 23.666666666666668,
 6.0,
 23.666666666666668,
 -23.0,
 -23.0,
 -23.0,
 23.666666666666668,
```

```
 -23.0,
 23.666666666666668,
 23.666666666666668,
 10.333333333333334,
 -5.0,
 23.666666666666668,
 23.666666666666668,
 23.666666666666668,
 23.666666666666668,
 -23.0,
 -23.0,
 -23.0,
 -21.0,
 -23.0,
 -23.0,
 -23.0,
 23.666666666666668,
 -23.0,
 -23.0,
 -23.0,
 23.666666666666668,
 -23.0,
 -23.0,
 -23.0,
 23.666666666666668,
 -23.0,
 23.666666666666668,
 23.666666666666668,
 -23.0,
 -23.0,
 -23.0,
 23.666666666666668,
 -23.0,
 -23.0,
 -23.0,
 23.666666666666668,
 1.3333333333333333,
 -23.0,
 -23.0,
 23.666666666666668,
 23.666666666666668,
 23.666666666666668,
 -23.0,
 -10.666666666666666]
```

Classification

In [14]:

```
knn_classification(X_train_class_shuffled, y_train_class_shuffled, shuffled_X_test_class, k)
```

Out[14]:

```
[1,
 0,
 1,
 1,
 0,
 1,
 0,
 0,
 1,
 0,
 0,
 0,
 0,
 0,
 0,
 0,
 0,
 0,
 0,
 0,
```

0,
0,
1,
0,
0,
1,
1,
1,
1,
1,
1,
0,
1,
1,
1,
1,
0,
0,
1,
0,
1,
1,
1,
0,
1,
1,
0,
0,
1,
0,
1,
0,
1,
0,
0,
0,
1,
0,
1,
1,
0,
0,
1,
1,
1,
1,
0,
0,
0,
0,
0,
0,
0,
0,
1,
0,
0,
0,
1,
0,
0,
0,
1,
0,
1,
1,
0,
0,
0,
1,
0,
0

```
0,
0,
1,
0,
0,
0,
1,
0,
1,
0,
0]
```

Apply custom implementations for multiple k values on pseudo-random data

In [15]:

```
k_values = [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Multiple K Regression Testing

In [16]:

```
regression_results_diff_k = {k: knn_regression(X_train_regress_shuffled, y_train_regress_
shuffled, shuffled_X_test_regress, k) for k in k_values}
```

Multiple K Classification Testing

In [17]:

```
classification_results_diff_k = {k: knn_classification(X_train_class_shuffled, y_train_cl
ass_shuffled, X_test_class, k) for k in k_values}
```

Built-In (Scikit-learn) Comparison Testing

In [18]:

```
from sklearn.neighbors import KNeighborsClassifier, KNeighborsRegressor
```

Regression Testing for Pseudo-Random Data

In [19]:

```
knn_regressor_pseudo = KNeighborsRegressor(n_neighbors=k)

knn_regressor_pseudo.fit(X_train_regress_shuffled, y_train_regress_shuffled)

knn_regressor_pseudo = knn_regressor_pseudo.predict(shuffled_X_test_regress)
```

Classification Testing for Pseudo-Random Data

In [20]:

```
# Classification for pseudo-random dataset
knn_classifier_pseudo = KNeighborsClassifier(n_neighbors=k)

knn_classifier_pseudo.fit(X_train_class_shuffled, y_train_class_shuffled)

pseudo_classification_result = knn_classifier_pseudo.predict(shuffled_X_test_class)
```

Real-World Testing

In [21]:

```
# Importing the Diabetes dataset from scikit-learn
from sklearn.datasets import load_diabetes
```

Load the Diabetes dataset

In [22]:

```
diabetes_data = load_diabetes()

X_real = diabetes_data.data.tolist()  # Features
y_real = diabetes_data.target.tolist()  # Target values
```

Create training and testing sets (80/20 split)

In [23]:

```
k = 7

split_index = int(len(X_real) * 0.8)

X_train_real, X_test_real = X_real[:split_index], X_real[split_index:]
```

Regression Testing Labels

In [24]:

```
y_train_real_regress, y_test_real_regress = y_real[:split_index], y_real[split_index:]
```

Classification Testing Labels

In [25]:

```
# Convert the target values to binary labels for classification (based on median value)
y_real_class = [1 if y > float(sum(y_real) / len(y_real)) else 0 for y in y_real]
```

In [26]:

```
y_train_real_class, y_test_real_class = y_real_class[:split_index], y_real_class[split_i
ndex:]
```

Regression Testing for Real-World Data

In [27]:

```
knn_regressor_real = KNeighborsRegressor(n_neighbors=k)

knn_regressor_real.fit(X_train_real, y_train_real_regress)

real_regression_result = knn_regressor_real.predict(X_test_real)
```

Classification Testing for Real-World Data

In [28]:

```
# Classification for real-world dataset
knn_classifier_real = KNeighborsClassifier(n_neighbors=k)

knn_classifier_real.fit(X_train_real, y_train_real_class)

real_classification_result = knn_classifier_real.predict(X_test_real)
```

"Manual" Implementations and Built-In Methods Comparison Using Multiple K Values

In [29]:

```
# Import metric computations for both types of KNNs (classification and regression)
from sklearn.metrics import confusion_matrix, mean_squared_error, precision_score, recall
_score, roc_auc_score
```

In [40]:

```
# Initialize variables to store results
mse_builtin_pseudo = []
mse_custom_pseudo = []
mse_custom_real = []
mse_builtin_real = []

confusion_builtin_pseudo = []
confusion_custom_pseudo = []
confusion_custom_real = []
confusion_builtin_real = []

recall_custom_pseudo = []
roc_auc_custom_pseudo = []
precision_custom_pseudo = []

recall_builtin_pseudo = []
roc_auc_builtin_pseudo = []
precision_builtin_pseudo = []

recall_custom_real = []
roc_auc_custom_real = []
precision_custom_real = []

recall_builtin_real = []
roc_auc_builtin_real = []
precision_builtin_real = []
```

Evaluation Loop through Different K Values

In [41]:

```
for k in k_values:
    print(k)
    # Custom and Built-in K-NN on Pseudo-random dataset
    pseudo_custom_class = knn_classification(X_train_class_shuffled, y_train_class_shuffl
ed, shuffled_X_test_class, k)
    pseudo_custom_regress = knn_regression(X_train_regress_shuffled, y_train_regress_shuf
fled, shuffled_X_test_regress, k)

    knn_classifier_k_pseudo = KNeighborsClassifier(n_neighbors=k)
    knn_regressor_k_pseudo = KNeighborsRegressor(n_neighbors=k)

    knn_classifier_k_pseudo.fit(X_train_class_shuffled, y_train_class_shuffled)
    knn_regressor_k_pseudo.fit(X_train_regress_shuffled, y_train_regress_shuffled)

    pseudo_builtin_class = knn_classifier_k_pseudo.predict(shuffled_X_test_class)
    pseudo_builtin_regress = knn_regressor_k_pseudo.predict(shuffled_X_test_regress)

    # Custom and Built-in K-NN on Real-world dataset
    real_custom_class = knn_classification(X_train_real, y_train_real_class, X_test_real
, k)
    real_custom_regress = knn_regression(X_train_real, y_train_real_regress, X_test_real
, k)

    knn_classifier_k_real = KNeighborsClassifier(n_neighbors=k)
    knn_regressor_k_real = KNeighborsRegressor(n_neighbors=k)

    knn_classifier_k_real.fit(X_train_real, y_train_real_class)
    knn_regressor_k_real.fit(X_train_real, y_train_real_regress)

    real_builtin_class = knn_classifier_k_real.predict(X_test_real)
```

```
    real_builtin_regress = knn_regressor_k_real.predict(X_test_real)

    # Calculate and store MSE for regression
    mse_builtin_pseudo.append(mean_squared_error(shuffled_y_test_regress, pseudo_builtin_
regress))
    mse_custom_pseudo.append(mean_squared_error(shuffled_y_test_regress, pseudo_custom_re
gress))
    mse_builtin_real.append(mean_squared_error(y_test_real_regress, real_builtin_regress)
)
    mse_custom_real.append(mean_squared_error(y_test_real_regress, real_custom_regress))

    # Calculate and store Confusion Matrix for classification
    confusion_builtin_pseudo.append(confusion_matrix(shuffled_y_test_class, pseudo_builti
n_class))
    confusion_custom_pseudo.append(confusion_matrix(shuffled_y_test_class, pseudo_custom_
class))
    confusion_builtin_real.append(confusion_matrix(y_test_real_class, real_builtin_class
))
    confusion_custom_real.append(confusion_matrix(y_test_real_class, real_custom_class))

    # Calculate and store additional metrics for classification
    recall_custom_pseudo.append(recall_score(shuffled_y_test_class, pseudo_custom_class)
)
    roc_auc_custom_pseudo.append(roc_auc_score(shuffled_y_test_class, pseudo_custom_class
))
    precision_custom_pseudo.append(precision_score(shuffled_y_test_class, pseudo_custom_c
lass))

    recall_builtin_pseudo.append(recall_score(shuffled_y_test_class, pseudo_builtin_clas
s))
    roc_auc_builtin_pseudo.append(roc_auc_score(shuffled_y_test_class, pseudo_builtin_cla
ss))
    precision_builtin_pseudo.append(precision_score(shuffled_y_test_class, pseudo_builtin
_class))

    recall_custom_real.append(recall_score(y_test_real_class, real_custom_class))
    roc_auc_custom_real.append(roc_auc_score(y_test_real_class, real_custom_class))
    precision_custom_real.append(precision_score(y_test_real_class, real_custom_class))

    recall_builtin_real.append(recall_score(y_test_real_class, real_builtin_class))
    roc_auc_builtin_real.append(roc_auc_score(y_test_real_class, real_builtin_class))
    precision_builtin_real.append(precision_score(y_test_real_class, real_builtin_class)
)

# Display MSE and Confusion Matrix for different k-values
print(mse_custom_real == mse_builtin_real,
    mse_custom_pseudo == mse_builtin_pseudo,
    # confusion_builtin_real, confusion_custom_real, confusion_custom_pseudo, confusion_b
uiltin_pseudo,
)

# Verify metrics for different k-values between dataset and model-type combinations
precision_custom_real == precision_builtin_real, recall_custom_real == recall_builtin_rea
l, roc_auc_custom_real == roc_auc_builtin_real,\
    precision_custom_pseudo == precision_builtin_pseudo, recall_custom_pseudo == recall_b
uiltin_pseudo, roc_auc_custom_pseudo == roc_auc_builtin_pseudo,\
        precision_custom_pseudo == precision_custom_real, recall_custom_pseudo == recall
_custom_real, roc_auc_custom_pseudo == roc_auc_custom_real,\
            precision_builtin_pseudo == precision_builtin_real, recall_builtin_pseudo ==
recall_builtin_real, roc_auc_builtin_pseudo == roc_auc_builtin_real
```

```
1
2
3
4
5
6
7
8
9
True False
```

Out[41]:

Out[41]:

```
(True, True, True, True, True, True, False, False, False, False, False, False)
```

Visualizations

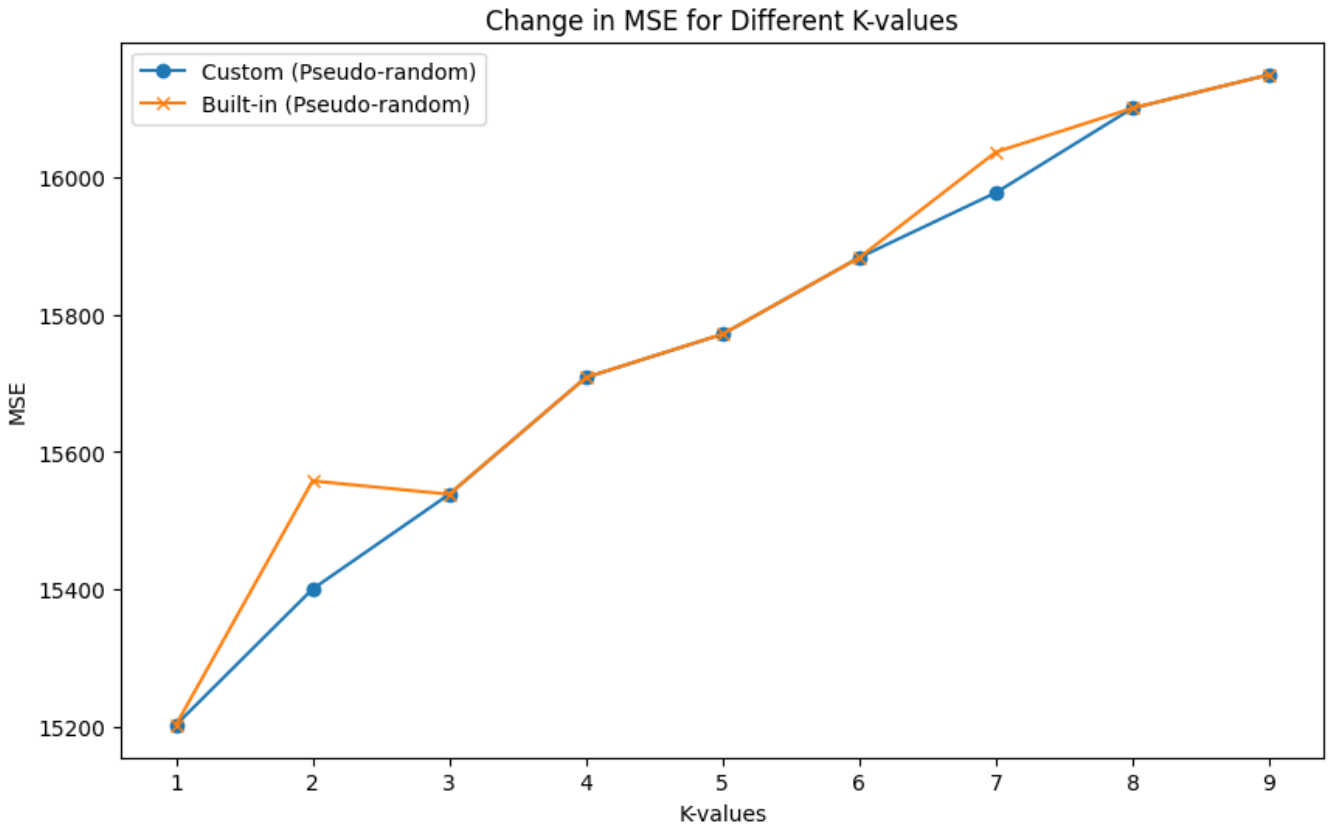
In [42]:

```
import matplotlib.pyplot as plt
```

Regression

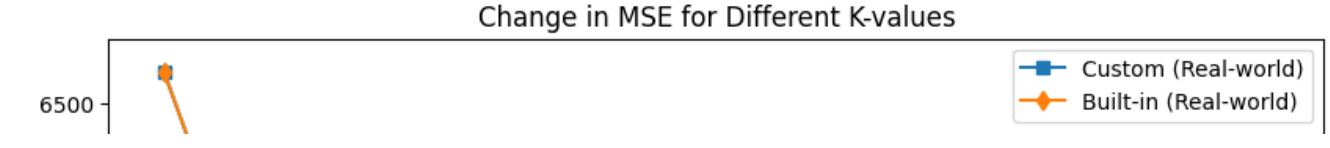
In [44]:

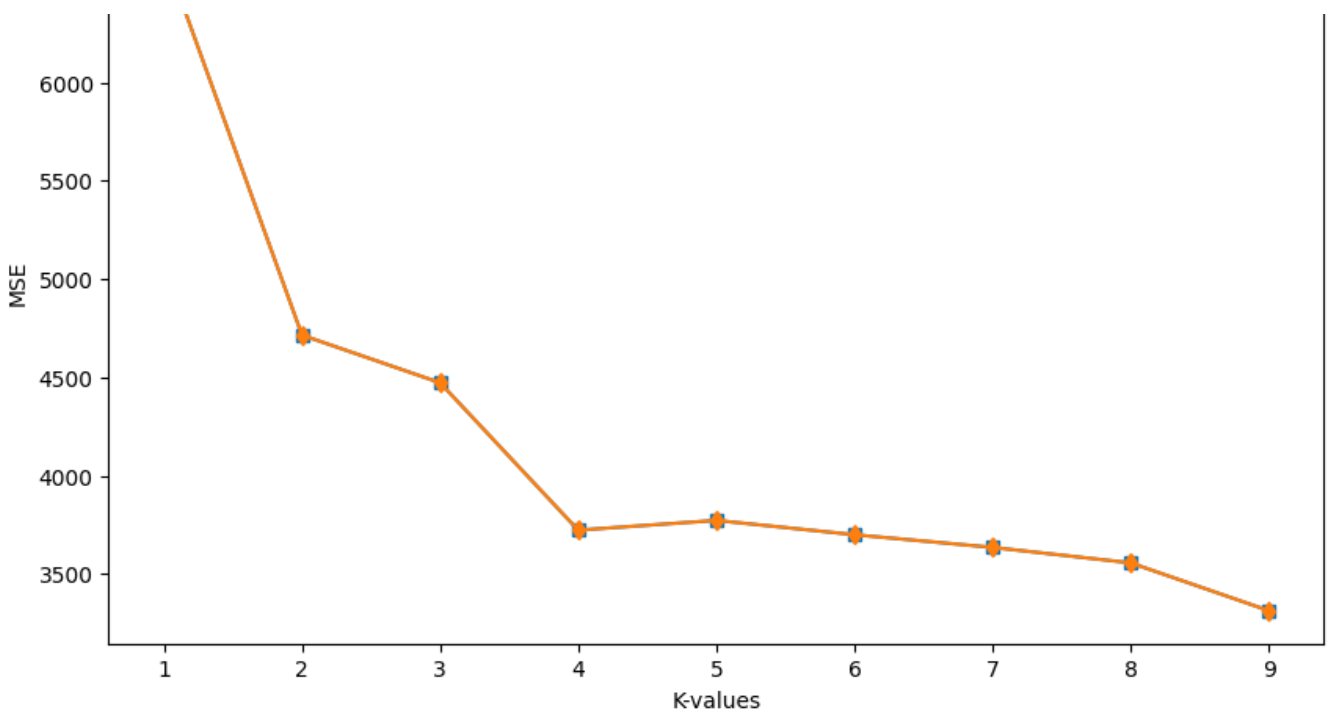
```
plt.figure(figsize=(10, 6))
plt.plot(k_values, mse_custom_pseudo, marker='o', label='Custom (Pseudo-random)')
plt.plot(k_values, mse_builtin_pseudo, marker='x', label='Built-in (Pseudo-random)')
plt.title('Change in MSE for Different K-values')
plt.xlabel('K-values')
plt.ylabel('MSE')
# plt.grid(True)
plt.legend()
plt.show()
```



In [45]:

```
plt.figure(figsize=(10, 6))
plt.plot(k_values, mse_custom_real, marker='s', label='Custom (Real-world)')
plt.plot(k_values, mse_builtin_real, marker='d', label='Built-in (Real-world)')
plt.title('Change in MSE for Different K-values')
plt.xlabel('K-values')
plt.ylabel('MSE')
plt.legend()
plt.show()
```





Classification

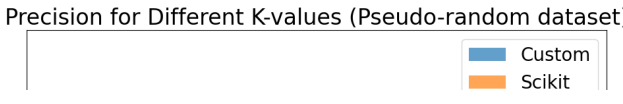
In [46]:

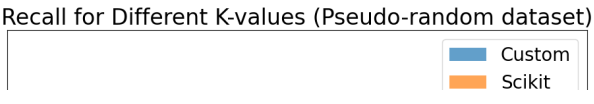
```
bar_width = 0.35
index = range(len(k_values))
```

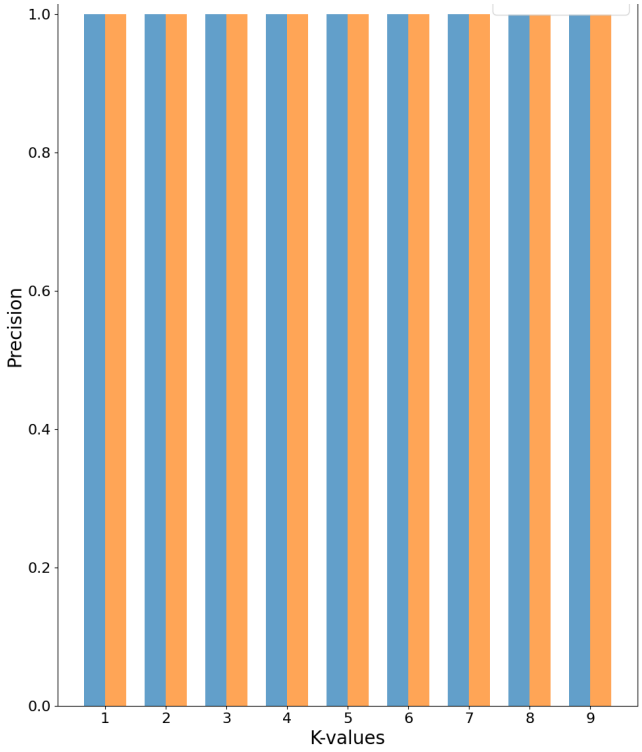
Pseudo-random Dataset

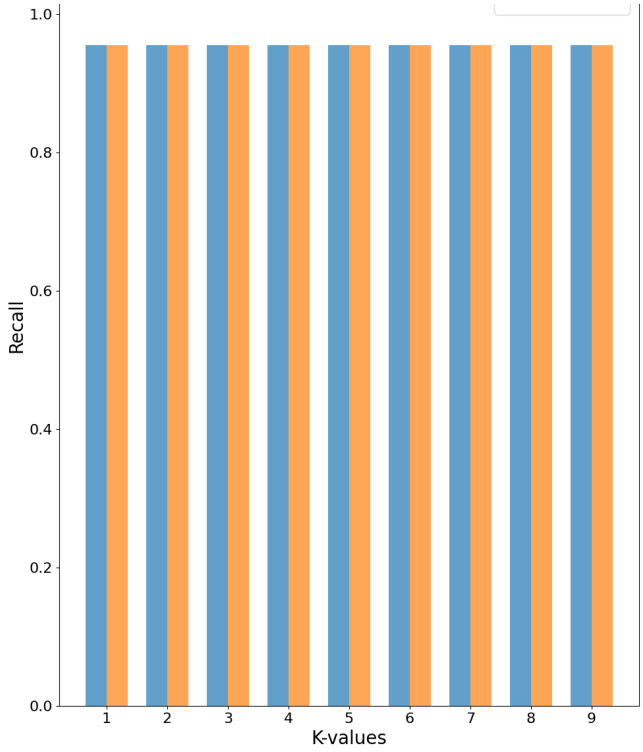
In [47]:

```
plt.figure(figsize=(25, 15))

plt.subplot(1, 2, 1)
plt.ylim(0, 1.1)
plt.bar(index, precision_custom_pseudo, bar_width, label='Custom', alpha=0.7)
plt.bar([i + bar_width for i in index], precision_builtin_pseudo, bar_width, label='Scik
it', alpha=0.7)
plt.xticks([i + bar_width / 2 for i in index], [str(k) for k in k_values], fontsize=16)
plt.title('Precision for Different K-values (Pseudo-random dataset)', fontsize=24)
plt.xlabel('K-values', fontsize=20)
plt.ylabel('Precision', fontsize=20)
plt.yticks(fontsize=16)
plt.xticks(fontsize=16)
plt.legend(fontsize=20)

plt.subplot(1, 2, 2)
plt.ylim(0, 1.1)
plt.bar(index, recall_custom_pseudo, bar_width, label='Custom', alpha=0.7)
plt.bar([i + bar_width for i in index], recall_builtin_pseudo, bar_width, label='Scikit'
, alpha=0.7)
plt.xticks([i + bar_width / 2 for i in index], [str(k) for k in k_values], fontsize=16)
plt.title('Recall for Different K-values (Pseudo-random dataset)', fontsize=24)
plt.xlabel('K-values', fontsize=20)
plt.ylabel('Recall', fontsize=20)
plt.yticks(fontsize=16)
plt.xticks(fontsize=16)
plt.legend(fontsize=20)

plt.show()
```





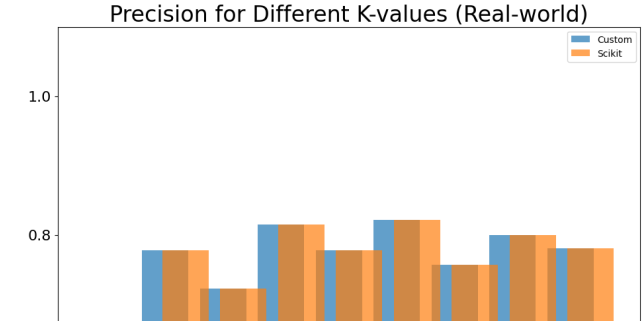


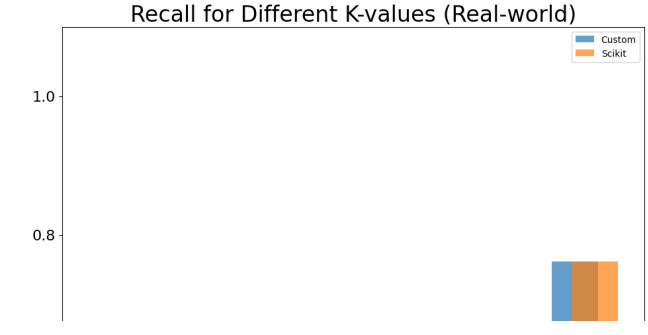


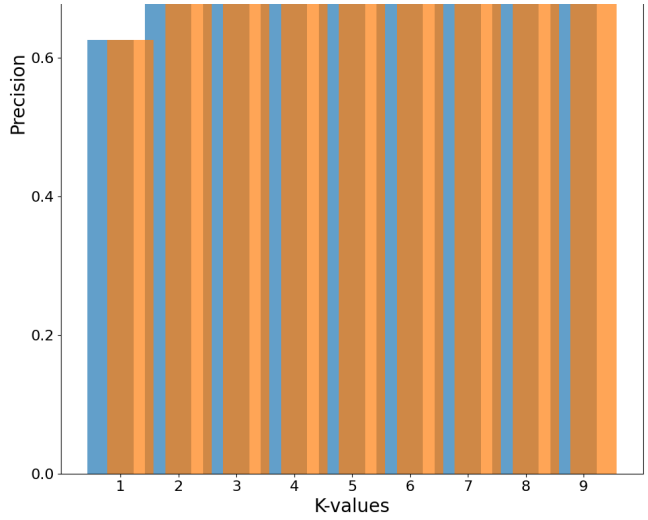
Real-world Dataset

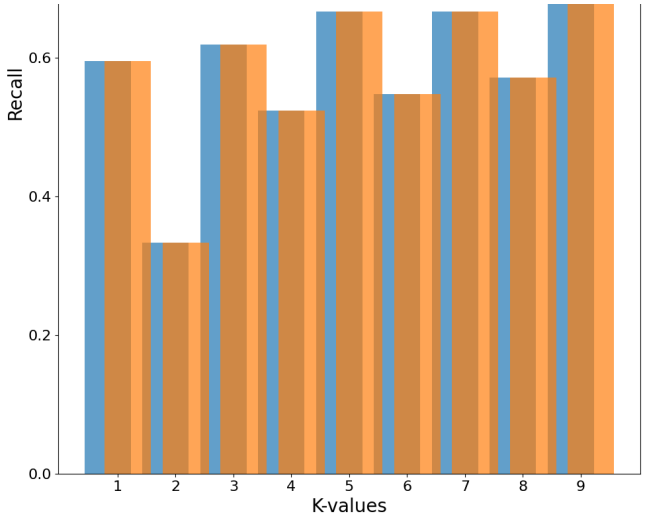
In [48]:

```
plt.figure(figsize=(25, 15))

plt.subplot(1, 2, 1)
plt.ylim(0, 1.1)
plt.bar(index, precision_custom_real, label='Custom', alpha=0.7)
plt.bar([i + bar_width for i in index], precision_builtin_real, label='Scikit', alpha=0.
7)
plt.xticks([i + bar_width / 2 for i in index], [str(k) for k in k_values], fontsize=16)
plt.title('Precision for Different K-values (Real-world)', fontsize=24)
plt.xlabel('K-values', fontsize=20)
plt.ylabel('Precision', fontsize=20)
plt.yticks(fontsize=16)
plt.xticks(fontsize=16)
plt.legend()

plt.subplot(1, 2, 2)
plt.ylim(0, 1.1)
plt.bar(index, recall_custom_real, label='Custom', alpha=0.7)
plt.bar([i + bar_width for i in index], recall_builtin_real, label='Scikit', alpha=0.7)
plt.xticks([i + bar_width / 2 for i in index], [str(k) for k in k_values], fontsize=16)
plt.title('Recall for Different K-values (Real-world)', fontsize=24)
plt.xlabel('K-values', fontsize=20)
plt.ylabel('Recall', fontsize=20)
plt.yticks(fontsize=16)
plt.xticks(fontsize=16)
plt.legend()

plt.show()
```









Gradient Optimization

In [49]:

```
import numpy as np
```

In [50]:

```
# Function for Gradient Descent for Part (e)
def gradient_descent_e(A, y, learning_rate=0.01, iterations=1000):
    x = np.random.rand(A.shape[1])  # Initialize x randomly
    for i in range(iterations):
        # Calculate the gradient
        grad = 2 * np.dot(A.T, np.dot(A, x) - y) + 2 * x

        # Update x using gradient descent
        x = x - learning_rate * grad

        # Optional: Compute the function value to check convergence
        f_val = np.linalg.norm(np.dot(A, x) - y)**2 + np.linalg.norm(x)**2

    return x, f_val

# Test Gradient Descent for Part (e)
A_e = np.array([[1, 2], [3, 4], [5, 6]])
y_e = np.array([1, 2, 3])
x_e, f_val_e = gradient_descent_e(A_e, y_e)
x_e, f_val_e
```

Out[50]:

```
(array([0.18965517, 0.34482759]), 0.17241379310344823)
```

In [51]:

```
# Modified Function for Gradient Descent for Part (c)
def gradient_descent_c(alpha, y, learning_rate=0.01, iterations=1000, epsilon=1e-8):
    x = np.random.rand(alpha.shape[0])  # Initialize x randomly
    f_val = None
    for i in range(iterations):
        dot_product = np.dot(alpha, x)

        # Adding safeguards for numerical stability
        dot_product = np.clip(dot_product, epsilon, 1 - epsilon)

        # Calculate the gradient
        grad = -y * alpha / dot_product + (1 - y) * alpha / (1 - dot_product)

        # Update x using gradient descent
        x = x - learning_rate * grad
```

```
        # Optional: Compute the function value to check convergence
        f_val = -y * np.log(dot_product) - (1 - y) * np.log(1 - dot_product)

    return x, f_val

# Test Gradient Descent for Part (c)
alpha_c = np.array([0.2, 0.4, 0.1])
y_c = 1
x_c, f_val_c = gradient_descent_c(alpha_c, y_c)
x_c, f_val_c
```

Out[51]:

```
(array([3.01620805, 4.97965565, 1.37501721]), 1.0000000100247594e-08)
```