In [91]:
```
# Problem 1
```

In [92]:
```
%matplotlib inline
from typing import Callable, Dict, List, Tuple, Union

import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import seaborn as sns
import statsmodels.formula.api as smf
import statsmodels.api as sm

from sklearn.linear_model import LinearRegression
from scipy.stats import binom
from scipy.stats import norm
from scipy.stats import ttest_ind
from tqdm import tqdm
from matplotlib import cm

sns.set(font_scale=1.5)
sns.set_style("whitegrid", {'grid.linestyle':'--'})
```

In [93]:
```
#import the data
data_1 = pd.read_csv("https://raw.githubusercontent.com/changyaochen/MECE4520/master/data/simple_linear_re

#create header row
col_Names_1=["X", "y"]
data_1 = pd.read_csv("https://raw.githubusercontent.com/changyaochen/MECE4520/master/data/simple_linear_re
data_1 #output the table
```

Out[93]:

	X	y
0	-15.000	-475.672
1	-14.824	-309.553
2	-14.648	-353.402
3	-14.472	-478.037
4	-14.296	-399.064
...	...	...
195	19.296	92.122
196	19.472	322.165
197	19.648	342.109
198	19.824	333.040
199	20.000	350.654

200 rows × 2 columns

In [94]:
```
# define the loss function
def linear_regression_loss(
    X: np.ndarray,
    y: np.ndarray,
    betas: np.ndarray,
    normalized: bool = True,
) -> float:
    """Calculates the loss of a linear regression problem.

    Args:
        X: The array (matrix) that contains the independent variables. By convention, the shape is n by (c
        y: The array (column vector) that contains the depedent variable. The shape is n by 1.
        betas: The array that contains the coefficients. The shape is (d+1) by 1.
        normalized: If True, the loss function is normalized by sample sizes (i.e., n). Default value: Fal

    Returns:
        (float): The loss function value.

    Raises:
        AssertionError: If the shape of the predicted value is different from y.
    """
    if not isinstance(betas, np.ndarray):
        betas = np.array(betas).reshape(-1, 1)
    if y.ndim == 1:
        y = y.reshape(-1, 1)

    y_pred = X @ betas
    assert y.shape == y_pred.shape, f"y has shape of {y.shape} and y_pred of {y_pred.shape}"
    loss = np.sum(np.square(y - X @ betas))

    if normalized:
        loss /= len(y)

    return loss
```

In [95]:
```
# print the value of the loss function
X_data_1 = np.vstack((np.ones(shape=len(data_1)), data_1["X"].values.T)).T
y_data_1 = data_1["y"].values

data_betas = [1,1]

print(f"The loss function is: {linear_regression_loss(X = X_data_1, y = y_data_1, betas = data_betas)}")
```

```
The loss function is: 29373.39868275
```

In [96]:
```
# define the gradient
def linear_regression_loss_gradient(
    X: np.ndarray,
    y: np.ndarray,
    betas: np.ndarray,
    normalized: bool = True,
) -> np.ndarray:
    """Calculates the gradient of the loss of a linear regression problem.

    Args:
        X: The array (matrix) that contains the independent variables. By convention, the shape is n by (
        y: The array (column vector) that contains the depedent variable. The shape is n by 1.
        betas: The array that contains the coefficients. The shape is (d+1) by 1.
        normalized: If True, the gradient is normalized by sample sizes (i.e., n). Default value: False.

    Returns:
        (float): The value of the gradient.
    """
    if not isinstance(betas, np.ndarray):
        betas = np.array(betas).reshape(-1, 1)
    if y.ndim == 1:
        y = y.reshape(-1, 1)

    grad = -2 * (X.T @ (y - X @ betas))
    assert grad.shape == betas.shape, f"The shape of grad is {grad.shape} and betas is {betas.shape}"
    if normalized:
        grad /= len(y)

    return grad
```

In [107]:
```
# define gradient descent
def gradient_descent(
    X: np.ndarray,
    y: np.ndarray,
    initial_guess: Union[List, np.ndarray],
    learning_rate: float,
    loss_function: Callable,
    gradient_function: Callable,
    verbose: bool = False,
    threshold: float = 1e-6,
    fix_guess: Dict = None,
) -> Tuple[List, List]:
    """Gradient descent routine.

    Args:
        X: The input data without labels.
        y: The labels.
        initial_guess: The starting point of the gradient descent.
        learning_rate: The learning rate of the gradient descent.
        loss_function: Provided loss function, that takes (X, y, parameter) as inputs.
        gradient_function: Provided gradient of the loss function, that takes (X, y, parameter) as inputs
        verbose: If set to True, print out intermediate results.
        threshold: Absolute value of different in loss, below which is considered converged.
        fix_guess: A dictionay to fix given dimension(s) of the parameter.

    Return:
        List: The history of parameters.
        List: The history of the losses.
    """
    guess_current = np.array(initial_guess).reshape(-1, 1)
    if fix_guess:
        for k, v in fix_guess.items():
            guess_current[k] = v

    guess_iter = [guess_current]
    losses_iter = [loss_function(X=X, y=y, betas=guess_current)]

    difference = float("inf")
    iteration_count = 0
    while abs(difference) > threshold:
        iteration_count += 1
        guess_next = guess_current - learning_rate * gradient_function(
            X=X, y=y, betas=guess_current)
```

```
        if fix_guess:
            for k, v in fix_guess.items():
                guess_next[k] = v
        guess_iter.append(guess_next)

        losses_next = loss_function(X=X, y=y, betas=guess_next)
        difference = losses_next - losses_iter[-1]
        losses_iter.append(losses_next)

        # update guess
        guess_current = guess_next
        # to print out intermediate results
        if verbose and iteration_count % 10 == 0:
            print(guess_next, losses_next)

    # some datatype processing
    guess_iter: List[List[float]] = list(map(lambda x: list(x.flatten()), guess_iter))
    return guess_iter, losses_iter
```

In [108]:

```
# print the gradient function
data_1_guess_iter, data_1_losses_iter = gradient_descent(
    X = X_data_1,
    y = y_data_1,
    initial_guess = [1, 1],
    learning_rate = 1e-3,
    loss_function = linear_regression_loss,
    gradient_function = linear_regression_loss_gradient,
    verbose = False,
);

print("The final values are:")
print(data_1_guess_iter[10])
```

```
The final values are:
[0.7720570767840739, 13.591579826304445]
```

In [109]:

```
# Problem 2
```

In [110]:
```
%matplotlib inline
from typing import Callable, Dict, List, Tuple, Union

import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import seaborn as sns
import statsmodels.formula.api as smf
import statsmodels.api as sm

from sklearn.linear_model import LinearRegression
from scipy.stats import binom
from scipy.stats import norm
from scipy.stats import ttest_ind
from tqdm import tqdm
from matplotlib import cm

sns.set(font_scale=1.5)
sns.set_style("whitegrid", {'grid.linestyle':'--'})
```

In [111]:
```
#import the data
data_2 = pd.read_csv("https://raw.githubusercontent.com/changyaochen/MECE4520/master/data/logistic_regres

#create header row
col_Names_2 = ["X1", "X2", "y"]
data_2 = pd.read_csv("https://raw.githubusercontent.com/changyaochen/MECE4520/master/data/logistic_regres
data_2 #output the table
```

Out[111]:

	X1	X2	y
0	1.250235	1.813271	1
1	3.342680	-2.721091	0
2	4.153036	1.776070	0
3	2.747564	-1.311193	0
4	3.981321	0.305327	0
...	...	...	...
195	3.649173	0.569156	0
196	2.825218	-6.146016	0
197	4.017264	2.351358	0
198	2.400017	-0.698954	0
199	4.576167	-3.553234	0

200 rows × 3 columns

In [112]:
```
# define logistic regression
def sigmoid(x):
    return 1 / (1 + np.exp(-x))

def logistic_regression_loss(
    X: np.ndarray,
    y: np.ndarray,
    betas: np.ndarray,
    normalized: bool = True,
) -> float:
    """Calculates the loss of a logistic regression problem, i.e., cross entropy."""
    if not isinstance(betas, np.ndarray):
        betas = np.array(betas).reshape(-1, 1)
    if y.ndim == 1:
        y = y.reshape(-1, 1)

    y_pred = X @ betas
    assert y.shape == y_pred.shape, f"y has shape of {y.shape} and y_pred of {y_pred.shape}"

    # We add a small positive number inside `log()` to avoid log(0)
    loss = np.sum(y * np.log(1e-10 + sigmoid(X @ betas)) + (1 - y) * np.log(1e-10 + 1 - sigmoid(X @ betas
    if normalized:
        loss /= len(y)

    return -1. * loss
```

In [113]:
```
# print the loss function
X_data_2 = np.vstack((np.ones(shape=len(data_2)), data_2["X1"].values.T, data_2["X2"].values.T)).T
y_data_2 = data_2["y"].values

data_betas = [5,-5,5]

print(f"The loss function is: {logistic_regression_loss(X = X_data_2, y = y_data_2, betas = data_betas)}"
```

```
The loss function is: 0.05093603628302528
```

In [114]:
```
# define the gradient
def logistic_regression_loss_gradient(
    X: np.ndarray,
    y: np.ndarray,
    betas: np.ndarray,
    normalized: bool = True,
) -> float:
    """Calculate the loss of a logistic regression problem, i.e., cross entropy."""
    if not isinstance(betas, np.ndarray):
        betas = np.array(betas).reshape(-1, 1)
    if y.ndim == 1:
        y = y.reshape(-1, 1)

    grad = -1. * (X.T @ (y - sigmoid(X @ betas)))
    assert grad.shape == betas.shape, f"The shape of grad is {grad.shape} and betas is {betas.shape}"
    if normalized:
        grad /= len(y)

    return grad
```

In [115]:
```
# gradient descent for logistic regression
data_2_guess_iter, data_2_losses_iter = gradient_descent(
    X = X_data_2,
    y = y_data_2,
    initial_guess = [5, -5, 5],
    learning_rate = 1,
    loss_function = logistic_regression_loss,
    gradient_function = logistic_regression_loss_gradient,
    threshold = 1e-7,
)

print("The final values are:")
print(data_2_guess_iter[2000])
```

```
The final values are:
[4.845584997499245, -6.617224052756929, 8.234758155828207]
```