Week 2 - Logistic Regresion & SVM
Machine Learning

David Esteso Calatrava

October 5, 2024

School of Computer Scinece and Statics

A. Logistic Regresion

i. Dataset visualization

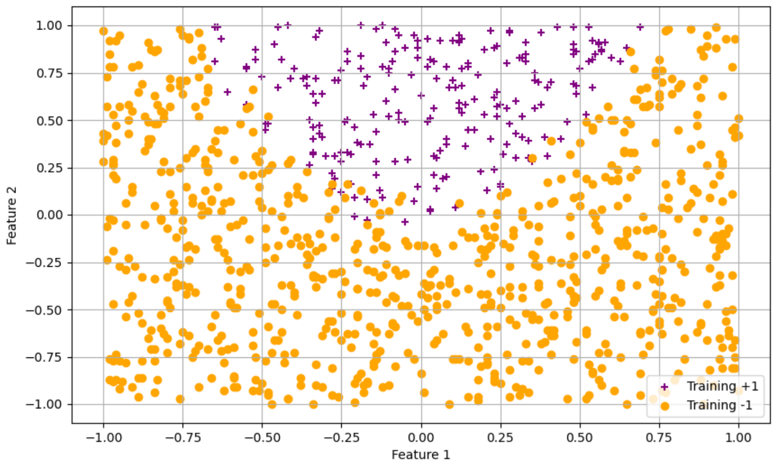


Figure 1: Dataset with id:1–2-1

Figure 1 displays pairs of feature values in 2D, with the x-axis representing the first feature and the y-axis representing the second. Points are marked with a purple plus for target value +1 and an orange circle for target value -1, illustrating the distribution of the data. By looking at Figure 1, we can make some guesses about how the model will behave. The points are mostly grouped in the upper area, which suggests that the **second feature** (on the y-axis) will have **more influence**. As the values of this feature go up, it seems like the model will predict class +1 more often. Because of this, we can expect the coefficient for the second feature to be **positive** and **large**. The **first feature** (on the x-axis) has a more **balanced spread**, so its influence might be **smaller**, and its coefficient could be lower.

As for the intercept, since there seem to be more points of class -1 overall, we might expect a **negative intercept** to reflect that class -1 is the more common outcome.

ii. Model features

An LR model was trained using the LR function provided by scikit-learn:

```
X = df.iloc[:, :features].values
y = df.iloc[:, features].values
# Train a logistic regression model
model = LogisticRegression().fit(X, y)
```

In this code, X contains the feature values taken from the first `features` columns of the DataFrame `df`, while y contains the target labels taken from the column indexed by `features`. A logistic regression model is trained using the feature data X and the target labels y.

Once a model is trained using a scikit-learn function, such as logistic regression or SVM, we can easily access various parameters using attributes of the model object. For instance, we can access to the coefficients or score as follows:

```
coefficients = model.coef_
intercept = model.intercept_
model_score = model.score(X, y)
```

Parameter	Value
Coefficients	[0.2267, 3.6083]
Intercept	-2.1505
Model Score	0.8287

Table 1: Logistic Regression Model Parameters

This section summarizes the key findings from the logistic regression model, findings that align with our expectations.

The first feature's coefficient is about **0.23**. This means that for every one-unit increase in this feature, we would expect the log odds of getting a positive outcome to increase by **0.23**. In simpler terms, as this feature's value goes up, it makes it more likely to predict the positive class ($y = +1$).

The second feature's coefficient is **3.61**, which is much larger. This indicates that a one-unit increase in this feature leads to a significant increase in the log odds. Essentially, this means that this feature has a **strong positive influence** on the prediction, making it much more likely that the outcome will be positive as its value increases.

The term "**log odds**" refers to the logarithm of the odds of an event happening. In logistic regression, instead of predicting probabilities directly, the model predicts the log odds, which makes it easier to model how the features influence the likelihood of a positive outcome.

A **negative intercept** suggests that, in the absence of other features, the model predicts a higher likelihood of the negative class ($y = -1$).

The model score is about **0.83**, meaning approximately 83% of the predictions matched the actual outcomes, which is considered a **good level of accuracy**.

$$\text{Model Score} = \frac{\text{Number of correct predictions}}{\text{Total number of samples}}$$

In summary, both features positively influence the prediction, with the second feature having a **much stronger effect**.

iii & iv. Model predictions

The decision boundary for the logistic regression model can be expressed as:

$$\theta_1 \cdot x_1 + \theta_2 \cdot x_2 + b = 0$$

Where: θ_1 and θ_2 are the coefficients for features x_1 and x_2, respectively, and b is the intercept. The equation is set to 0 because it represents the threshold where the predicted probabilities of the classes are equal. This means that on the decision boundary, the model is indifferent between classifying a point as either class, effectively separating the two classes in the feature space. Substituting the specific values from the trained model and rearranging, we have the ecuation of the decision boundary:

$$x_2 = \frac{2.15 - 0.23 \cdot x_1}{3.61}$$

So, what can we expect about this boundary without seeing the plot?

- If the expression evaluates to greater than zero:

  $$0.23 \cdot \theta_1 + 3.61 \cdot \theta_2 - 2.15 > 0$$

  the point is classified as belonging to the **positive class** ($y = +1$), which would be situated in the **upper region** of the graph.

- Conversely, if the expression is less than zero:

  $$0.23 \cdot \theta_1 + 3.61 \cdot \theta_2 - 2.15 < 0$$

  the point is classified as belonging to the **negative class** ($y = -1$), which would be located in the **lower region** of the graph.

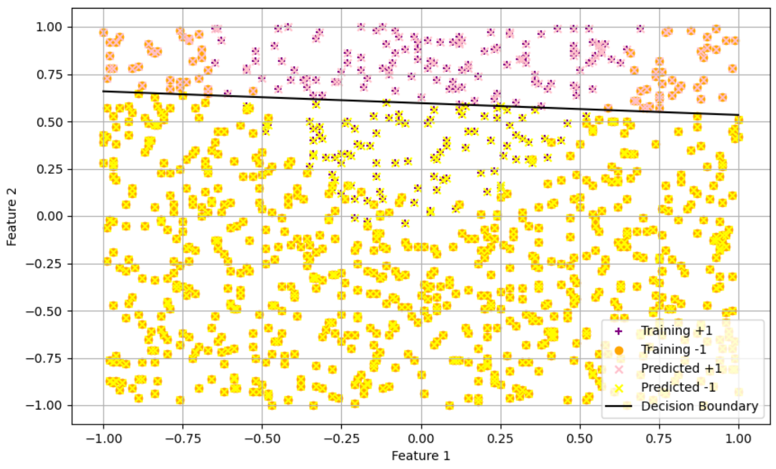
- The **negative coefficient** for x_1 in the rearranged equation indicates a **negative slope**.

The following code uses the trained `model` to predict outcomes based on the feature values extracted from the DataFrame `df`.

```
predictions = model.predict(df.iloc[:, :features].values)
```

Additionally, this code generates 100 `x_values` and computes the corresponding `y_values` using the decision boundary equation. It then plots the boundary as a black line labeled `Decision Boundary`.

```
x_values = np.linspace(X1.min(), X1.max(), 100)
# Solve for y values using the decision boundary equation: c + m1*x + m2*y = 0
y_values = -(c + m[0] * x_values) / m[1]
plt.plot(x_values, y_values, color='black', label='Decision␣Boundary')
```



In Figure 2, we confirm that our assumptions were correct, as the plot aligns with our expectations regarding the decision boundary and classified data. The model produces a decision boundary consistent with what we would expect from a linear model, effectively separating the classes. The class +1 predictions are represented by a pink cross, and the class -1 predictions by a yellow one.

B. SVM

iii. Impact of C in SVM

The SVM method aims to find the best boundary that separates different classes in the data. It works by looking for the "**maximum margin**" between the classes. The parameter C plays a crucial role in **balancing** the model's classification accuracy and the maximization of the margin. It controls the misclassification penalty during training, enabling a compromise between having a wider margin and achieving accurate classifications.

Understanding the Margin

The margin in SVM is the distance between the decision boundary and the closest data points from each class, with the goal of maximizing this margin while minimizing classification errors. A small value of C emphasizes maximizing the margin over correctly classifying all training examples, allowing for more **misclassifications**, which can be beneficial in noisy or overlapping data by reducing overfitting and improving generalization. Conversely, a large value of C prioritizes correct classification of all training examples, potentially leading to a narrower margin and **overfitting** if the data is not well-separated, making the model sensitive to individual data points and adversely affecting generalization to unseen data.

Understanding SVM with Loss Function and Gradient Descent

The cost function for Support Vector Machines (SVM) is defined as:

$$J(\theta) = \frac{1}{m}\sum_{i=1}^{m} \max(0, 1 - y^{(i)}(\theta^T x^{(i)})) + \frac{1}{C}\theta^T\theta$$

Breaking down the terms:

- $\frac{1}{m}\sum_{i=1}^{m}$: This term averages the hinge loss over all training examples, ensuring the cost function scales properly with the size of the dataset.
- $\max(0, 1 - y^{(i)}(\theta^T x^{(i)}))$: This is the hinge loss. It measures classification error.
  - If $y^{(i)}(\theta^T x^{(i)}) \geq 1$, the classification is correct, and the loss is 0.
  - If $y^{(i)}(\theta^T x^{(i)}) < 1$, the classification is incorrect or close to the margin, and the hinge loss increases.
- $\frac{1}{C}\theta^T\theta$: This is the regularization term, which influences the size of the parameters θ that minimize the cost function. $\theta^T\theta$ is the sum of the squares of the parameters, measuring the "size" of θ.

A small value of C encourages smaller parameter values, leading to a simpler model with smaller θ, which may allow more misclassifications but helps prevent overfitting by penalizing the model when parameters become too large, as represented by $\theta^T\theta$ (the sum of their squares). Conversely, a large value of C reduces the emphasis on this penalty term, allowing larger parameter values that can potentially increase accuracy on the training data while also increasing the risk of overfitting.

In the gradient descent optimization process, the parameters θ are updated based on the gradient of the cost function:

$$\theta_j := \theta_j - \alpha\left(\frac{\theta_j}{C} - \frac{1}{m}\sum_{i=1}^{m} y^{(i)} x_j^{(i)} \cdot 1\{y^{(i)}(\theta^T x^{(i)}) \leq 1\}\right)$$

Here, α is the learning rate, controlling the size of the updates. While C does not directly affect the learning rate, it influences the gradient by determining the penalty for large parameter values, thereby encouraging smaller θ values. A smaller C encourages less complex models by allowing larger parameter updates, while a larger C enforces stricter penalties, leading to smaller, more cautious updates.

i & ii. Training a model with differnet values of C

This code trains a linear SVM model for each value of C, using the SVM function given by scikitLearn:

```
for C in C_values:
    # Train a linear SVM model
    model = LinearSVC(C=C)
    model.fit(X, y)
```

Regularization (C)	Coefficients	Intercept	Iterations	Model Score
0.001	[0.02097372, 0.33632045]	-0.3668	10	0.778
0.01	[0.06255884, 0.82897717]	-0.5590	11	0.832
1	[0.09688126, 1.29317338]	-0.7446	66	0.831
10	[0.09742611, 1.30266602]	-0.7488	570	0.831

Table 2: SVM Model Parameters for Different Values of C

At **C** = **0.001**, the coefficients are low [0.02097372,0.33632045] and the model score is 0.778, indicating minimal feature influence and potential misclassification. What is more, as whe can see in Figure 2, this model equals to a baseline predictor.

Increasing **C** to **0.01** results in higher coefficients [0.06255884,0.82897717] and a model score of 0.832. This shift reflects a better balance between complexity and accuracy, as the intercept approaches zero, aligning with our earlier discussion about the flexibility afforded by a smaller **C**.

At **C** = **1**, the coefficients increase to [0.09688126,1.29317338] with a score of 0.831, indicating that optimal performance may have been achieved earlier, echoing the importance of finding the right balance as we discussed.

Finally, at **C** = **10**, coefficients are slightly higher [0.09742611,1.30266602], but the model requires significantly more iterations to converge (570 iterations), while the score remains at 0.831. This illustrates the risk of **overfitting**, as the increase in complexity does not translate into improved performance, highlighting our earlier emphasis on the necessity of selecting an appropriate **C** value to avoid inefficiency.

In summary, higher **C** values usually increase the coefficients and boost feature influence, but performance levels off after some value of **C**, in our case **C** = **0.01**. This shows the importance of balancing regularization to avoid **overfitting**, as we discussed earlier.

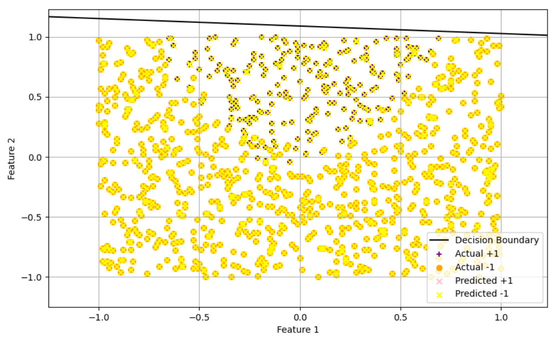


Figure 2: SVM with C=0.001

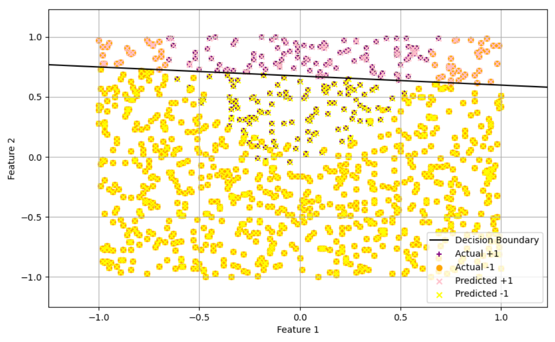


Figure 3: SVM with C=0.01

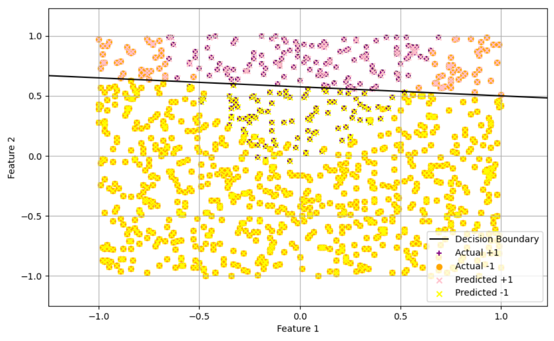


Figure 4: SVM with C=1

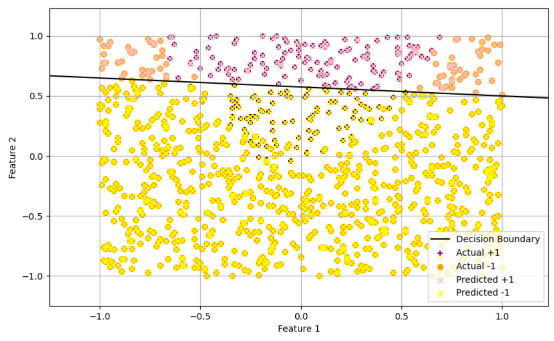


Figure 5: SVM with C=10

iv. Comparison of SVM and Logistic Regression Models

Now, we compare two **linear models**: Support Vector Machines (SVM) and logistic regression, both of which aim to find a linear decision boundary for classifying data. Despite both models achieving acceptable predictive accuracy, they share an **inherent limitation** due to their linear nature, which caps their potential in capturing more complex patterns in the data.

Both models use coefficients to inform predictions, but their **interpretations differ**. In logistic regression, the coefficients represent the **log odds** of the positive outcome, providing a probabilistic framework for classification. In contrast, the SVM coefficients indicate the **influence of features** on the decision boundary, focusing on **maximizing the margin** between classes. This often results in larger coefficients for SVM, reflecting its emphasis on maintaining distance from the decision boundary.

The **intercepts** for both models tend to be negative, suggesting a low likelihood of predicting the positive class ($y = 1$) in the absence of feature influence. However, the specific values vary due to the **distinct methodologies** each model employs in parameter estimation; for instance, logistic regression focuses on the overall likelihood of the dataset, while SVM adjusts its intercept based on maximizing the margin defined by support vectors.

In terms of model performance, the difference is **minimal**; logistic regression achieved an approximate score of 0.83, while SVM scores fluctuated with different values of the regularization parameter C, peaking at 0.832. This indicates that while both models yield similar results, reaching the performance of logistic regression required more extensive tuning of C in SVM, **making logistic regression a more straightforward choice for our case.**

C. Squared Features Logistic Regression Model

i & ii. Evaluating Non-linear Models vs. Linear SVM and Logistic Regression

The following code generates squared features from the original features and saves the new DataFrame to a CSV file:

```
X1 = df.iloc[:, 0]
X2 = df.iloc[:, 1]

# Generate squared features
X1_squared = X1 ** 2
X2_squared = X2 ** 2

# Create a new feature DataFrame
```

```
squared_features_df = pd.DataFrame({
    'Feature␣1': X1,
    'Feature␣2': X2,
    'Feature␣1␣Squared': X1_squared,
    'Feature␣2␣Squared': X2_squared,
    'Target': df.iloc[:, 2]
})

squared_features_df.to_csv(output_file, index=False)
```

In this code, `X1` and `X2` are extracted from the DataFrame `df`. Squared features are generated, and a new DataFrame `squared_features_df` is created containing the original features, their squares, and the target variable. Finally, this DataFrame is saved to a CSV file specified by `output_file`. The model has been trained like the logistic regression model in section A.

Parameter	**LR Model**	**SVM Model**	**Squared Features LR model**
Coefficients	[0.2267, 3.6083]	[0.0626, 0.8290]	[0.1438, 5.5103, -8.0270, -0.2957]
Intercept	-2.1505	-0.5590	-0.6780
Iterations	8	11	11
Model Score	0.8287	0.8318	0.9650
Regularization (C)	1.0	0.01	1.0

Table 3: Comparison of Logistic Regression and SVM Models

In this section, we compare the **initial logistic regression model** and the **SVM model** against the **model with squared features**.

Looking at the **coefficients**, the basic logistic regression model and the SVM model show simpler patterns in the data. The model with squared features, on the other hand, has four coefficients instead of two. This gives the model more information, allowing it to capture more details and patterns that the simpler models miss. This often leads to better predictions.

For the **intercept**, the squared features model keeps the same sign as the simpler models. The change in size could be because of the extra features added to the model, which help it better understand the data.

The **model scores** reveal a clear distinction in performance. The squared features model significantly outperforms both the initial logistic regression and the SVM models, demonstrating its ability to fit the data more effectively. This enhanced performance indicates that the squared features provide substantial additional information that improves the model's accuracy.

In summary, the squared features model represents a **significant advancement** over both the initial logistic regression and the SVM models. By capturing more complex relationships within the data, it achieves superior predictive performance and demonstrates the value of incorporating non-linear terms into the model.

iii. Baseline predictor comparisson

The following Python code identifies the most common class in the target variable and calculates its accuracy score:

```
target = df.iloc[:, -1]

most_frequent_class = target.mode()[0]
most_frequent_count = target.value_counts()[most_frequent_class]

accuracy_score = most_frequent_count / len(target)
```

In this snippet, we first extract the target variable from the last column of the DataFrame `df`. The `mode` function is used to find the most frequently occurring class. Next, we count how many times this class appears using `value_counts`. Finally, we compute the accuracy score by dividing the count of the most frequent class by the total number of instances in the target variable.

We compare three classifiers: a linear regression model and an SVM, both with an accuracy of 83%, and a linear regression model with squared features that achieves 97% accuracy, against a baseline predictor that always predicts the most common class, which has an accuracy of 78%. This comparison is especially important in the context of **imbalanced datasets**, where relying on accuracy can be misleading. While all **three classifiers exceed the baseline**, the squared features model stands out, emphasizing the need for careful model selection and evaluation metrics. In our dataset, although it is not extremely imbalanced, **one class represents over** 75% of the data, highlighting the significance of considering this aspect during evaluation.

However, we are not following **good practices** in our model evaluation. We are training and predicting on the same dataset, which can lead to overfitting and give an overly optimistic view of performance. We also haven't considered important metrics like recall, F1 score, or AUC, which are crucial for assessing model effectiveness, particularly in identifying the minority class. Adopting proper evaluation methods is vital for building reliable models.

iv. Decision Boundary of the Logistic Regression Model with Quadratic Features

For a model with quadratic features, the decision function is expressed as:

$$0 = b + \theta_1 x_1 + \theta_2 x_2 + \theta_3 x_1^2 + \theta_4 x_2^2$$

As previously mentioned, the decision boundary is found by setting $z = 0$. This equation describes the decision boundary in the feature space. For the given data and equation:

$$-0.2957 \cdot x^2 - 0.2957 \cdot y^2 = -5.5103 \cdot x + 8.0270 \cdot y - 0.1438$$

The negative values of $\theta_3 = -8.0270$ **and** $\theta_4 = -0.2957$ have an important effect on the shape of the decision boundary. These negative numbers make the boundary curve inward, creating an elliptical shape that separates the two classes.

The larger value of θ_3 (in absolute terms) causes a **sharper curve** along the x-axis. This is needed because the points along the x-axis have more variation between the positive and negative classes. So, the model needs more flexibility in this direction.

On the other hand, the smaller value of θ_4 leads to a gentler curve along the y-axis. The points along the y-axis are more spread out, so the model doesn't need such a strong curve to separate them.

This combination of sharper and smoother curves helps the model create a decision boundary that fits the data well, as Figure 6 evidences.

The following Python code generates a meshgrid and calculates the decision boundary:

```
xx1, xx2 = np.meshgrid(np.linspace(X1.min() + 0.25, X1.max() + 0.1, 100),
                       np.linspace(X2.min() + 0.25, X2.max() + 0.25, 100))

decision_boundary =
(m[0] * xx1 + m[1] * xx2 + c + m[2] * (xx1 ** 2) + m[3] * (xx2 ** 2))
plt.contour(xx1, xx2, decision_boundary, levels=[0], colors='black')
```

Here, a meshgrid `xx1` and `xx2` is created to cover the feature space defined by `X1` and `X2`. The decision boundary is calculated using the specified equation, combining linear and quadratic terms. Finally, the decision boundary is plotted using a contour line at the level 0.

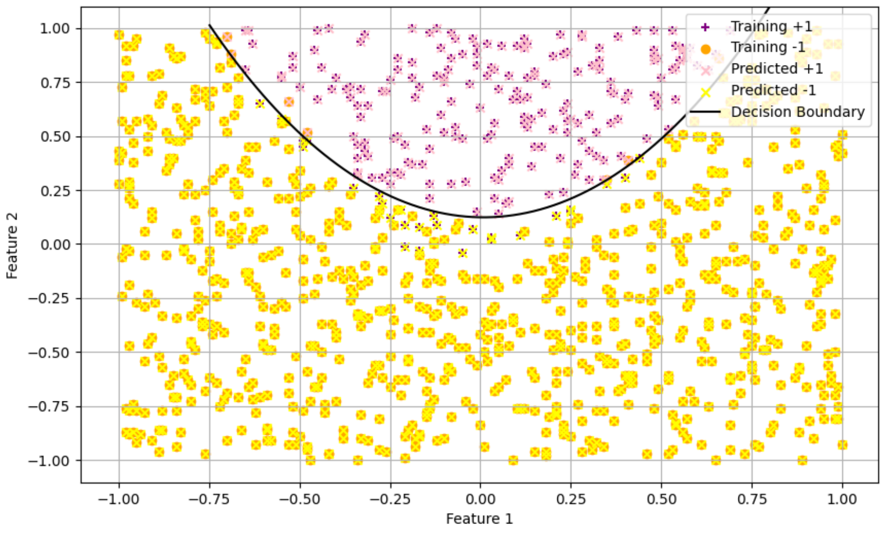


Figure 6: Decision boundary and predictions for Squared features model

Appendix

exercise_a.py

```python
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.linear_model import LogisticRegression

def load_data(filepath):
    """Load data from a CSV file."""
    df = pd.read_csv(filepath)
    return df

def visualize_data(df):
    """Visualize the data in a 2D plot."""
    X1 = df.iloc[:, 0]
    X2 = df.iloc[:, 1]
    y = df.iloc[:, 2]

    plt.figure(figsize=(10, 6))
    plt.scatter(X1[y == 1], X2[y == 1], marker='+', color='purple', label='Training␣+1')
    plt.scatter(X1[y == -1], X2[y == -1], marker='o', color='orange', label='Training␣-1')

    plt.xlabel('Feature␣1')
    plt.ylabel('Feature␣2')
    plt.legend()
    plt.grid(True)

    plt.savefig('../output/a/exercise_i.png')
    plt.close()

def visualize_predictions_with_decision_boundary(df, model):
    """Visualize training data, predictions, and decision boundary of the model."""
    features = df.shape[1] - 1
    X1 = df.iloc[:, 0].values
    X2 = df.iloc[:, 1].values
    y = df.iloc[:, features].values

    # Make predictions using the model
    predictions = model.predict(df.iloc[:, :features].values)
    c = model.intercept_[0]
    m = model.coef_[0]

    plt.figure(figsize=(10, 6))
    plt.scatter(X1[y == 1], X2[y == 1], marker='+', color='purple', label='Training␣+1')
    plt.scatter(X1[y == -1], X2[y == -1], marker='o', color='orange', label='Training␣-1')
    plt.scatter(X1[predictions == 1], X2[predictions == 1], marker='x', color='pink', label=
        'Predicted␣+1')
    plt.scatter(X1[predictions == -1], X2[predictions == -1], marker='x', color='yellow',
        label='Predicted␣-1')

    if features == 2:
        x_values = np.linspace(X1.min(), X1.max(), 100)
        # Solve for y values using the decision boundary equation: c + m1*x + m2*y = 0
        y_values = -(c + m[0] * x_values) / m[1]
        plt.plot(x_values, y_values, color='black', label='Decision␣Boundary')
    elif features == 4:
        xx1, xx2 = np.meshgrid(np.linspace(X1.min() + 0.25, X1.max() + 0.1, 100), np.
            linspace(X2.min() + 0.25, X2.max() + 0.25, 100))
        # Decision boundary equation: w0 + w1*x1 + w2*x2 + w3*x1^2 + w4*x2^2 = 0
        decision_boundary = (m[0] * xx1 + m[1] * xx2 + c + m[2] * (xx1 ** 2) + m[3] * (xx2
            ** 2))
        plt.contour(xx1, xx2, decision_boundary, levels=[0], colors='black')
        # Auxiliar line in order to represent the decision boundary in the legend (contour
            plot does not have a legend)
        plt.plot([], [], 'k-', label='Decision␣Boundary', color='black')

    plt.xlabel('Feature␣1')
    plt.ylabel('Feature␣2')
    plt.legend()
    plt.grid(True)

    filename = '../output/a/exercise_iii.png' if features == 2 else '../output/c/exercise_ii
        .png'
    plt.savefig(filename)
    plt.close()

def train_logistic_regression(df):
    """Train a logistic regression model and save feature influence analysis."""
    features = df.shape[1] - 1
    X = df.iloc[:, :features].values
    y = df.iloc[:, features].values

    # Train a logistic regression model
    model = LogisticRegression().fit(X, y)

    coefficients = model.coef_[0]
    intercept = model.intercept_[0]

    filename = '../output/a/exercise_ii.txt' if features == 2 else '../output/c/exercise_i.
        txt'
    with open(filename, 'w') as f:
        f.write("Logistic␣Regression␣Model␣Parameters:\n")
        f.write(f"Coefficients:␣{coefficients}\n")
        f.write(f"Intercept:␣{intercept}\n\n")
        f.write("Additional␣Model␣Information:\n")
        f.write(f"Number␣of␣Iterations:␣{model.n_iter_}\n")
        f.write(f"Model␣Score:␣{model.score(X,␣y)}\n")

    return model
```

exercise_b.py

```python
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.linear_model import LogisticRegression
from sklearn.svm import LinearSVC

def train_linear_svm(df, C_values):
    """Train linear SVM classifiers on data from a DataFrame for a range of C values."""
    X = df.iloc[:, :2].values
    y = df.iloc[:, 2].values
    models = []

    # Train a linear SVM model for each value of C
    with open('../output/b/exercise_i.txt', 'w') as f:
        for C in C_values:
            # Train a linear SVM model
            model = LinearSVC(C=C)
            model.fit(X, y)
            models.append(model)

            coefficients = model.coef_[0]
            intercept = model.intercept_[0]

            f.write(f"SVM␣Model␣with␣C={C}:\n")
            f.write(f"Coefficients:␣{coefficients}\n")
            f.write(f"Intercept:␣{intercept}\n\n")
            f.write(f"Number␣of␣Iterations:␣{model.n_iter_}\n")
            f.write(f"Model␣Score:␣{model.score(X,␣y)}\n")
            f.write(f"Regularization␣(C):␣{C}\n")
            f.write(f"\n\n")

    return models

def plot_svm_predictions(df, models, C_values):
    """Plot actual vs predicted values with decision boundaries for multiple SVM models."""
    X = df.iloc[:, :2].values
    y = df.iloc[:, 2].values

    for i, model in enumerate(models):
        # Make predictions using the model
        predictions = model.predict(X)

        x_min, x_max = X[:, 0].min() - 0.25, X[:, 0].max() + 0.25
        y_min, y_max = X[:, 1].min() - 0.25, X[:, 1].max() + 0.25
        xx, yy = np.meshgrid(np.arange(x_min, x_max, 0.02), np.arange(y_min, y_max, 0.02))

        c = model.intercept_[0]
        m = model.coef_[0]
        x_values = np.linspace(x_min, x_max, 100)
        # Solve for y values using the decision boundary equation: c + m1*x + m2*y = 0
        y_values = -(c + m[0] * x_values) / m[1]

        plt.figure(figsize=(10, 6))
        plt.plot(x_values, y_values, color='black', label='Decision␣Boundary')
        plt.scatter(X[:, 0], X[:, 1], c=y, edgecolors='k', marker='o', s=20)
        plt.scatter(X[y == 1, 0], X[y == 1, 1], marker='+', color='purple', label='Actual␣+1
            ')
        plt.scatter(X[y == -1, 0], X[y == -1, 1], marker='o', color='orange', label='Actual␣
            -1')
        plt.scatter(X[predictions == 1, 0], X[predictions == 1, 1], marker='x', color='pink'
            , label='Predicted␣+1')
        plt.scatter(X[predictions == -1, 0], X[predictions == -1, 1], marker='x', color='
            yellow', label='Predicted␣-1')

        plt.xlabel('Feature␣1')
        plt.ylabel('Feature␣2')
        plt.xlim(xx.min(), xx.max())
        plt.ylim(yy.min(), yy.max())
        plt.legend(loc='best')
        plt.grid(True)

        # Save each plot in a separate file
        plt.savefig(f'../output/b/exercise_ii_C_{C_values[i]}.png')
        plt.close()
```

exercise_c.py

```python
import pandas as pd
import exercise_a as a

def add_squared_features(df, output_file='../data/week2_squared.csv'):
    """Add squared features to the DataFrame and save to a CSV file."""
    X1 = df.iloc[:, 0]
    X2 = df.iloc[:, 1]

    # Generate squared features
    X1_squared = X1 ** 2
    X2_squared = X2 ** 2

    # Create a new feature DataFrame
    squared_features_df = pd.DataFrame({
        'Feature␣1': X1,
        'Feature␣2': X2,
        'Feature␣1␣Squared': X1_squared,
        'Feature␣2␣Squared': X2_squared,
        'Target': df.iloc[:, 2]
    })

    squared_features_df.to_csv(output_file, index=False)

def baseline_predictor(df, output_file='../output/c/exercise_iii.txt'):
    """Train a baseline model that always predicts the most frequent class."""
    target = df.iloc[:, -1]

    most_frequent_class = target.mode()[0]
    most_frequent_count = target.value_counts()[most_frequent_class]

    accuracy_score = most_frequent_count / len(target)

    with open(output_file, 'w') as f:
        f.write(f'Accuracy␣Score:␣{accuracy_score:.4f}\n')
```

main.py

```python
import exercise_a as a
import exercise_b as b
import exercise_c as c

if __name__ == "__main__":
    # Load the data
    data_filepath = '../data/week2.csv'
    df = a.load_data(data_filepath)

    # To solve exercise A
    a.visualize_data(df)
    model = a.train_logistic_regression(df)
    a.visualize_predictions_with_decision_boundary(df, model)

    # To solve exercise B
    C_values = [0.001, 0.01, 1, 10]
    models = b.train_linear_svm(df, C_values)
    b.plot_svm_predictions(df, models, C_values)

    # To solve exercise C
    c.add_squared_features(df)
    df = a.load_data('../data/week2_squared.csv')
    model = a.train_logistic_regression(df)
    a.visualize_predictions_with_decision_boundary(df, model)
    c.baseline_predictor(df)
```