Assignment 7
White Box Testing & Black Box Testing Report

Aditya Srivastava (2201013)
Deepanika Gupta (2201062)
Kartik Saini (2201103)

Contents

1 Introduction **2**
1.1 Testing Objectives . . . 2

2 White Box Testing **2**
2.1 Backend White Box Test Cases . . . 2
2.1.1 Analyzer Classes Initialization Test . . . 2
2.1.2 Language Detection and Translation Test . . . 3
2.1.3 Tweet Analysis Pipeline Test . . . 3
2.1.4 API Endpoint Security Test . . . 4
2.2 Frontend White Box Test Cases . . . 4
2.2.1 React Component State Management Test . . . 4
2.2.2 Route Navigation and Authentication Test . . . 5
2.2.3 API Error Handling Test . . . 6
2.2.4 TweetCard Component Rendering Test . . . 7
2.2.5 Authentication Flow Test . . . 8

3 Black Box Testing **9**
3.1 Frontend Black Box Test Cases . . . 9
3.1.1 User Registration Test . . . 9
3.1.2 Tweet Analysis with Different Parameters Test . . . 9
3.1.3 Search History Retrieval Test . . . 10
3.1.4 Navigation Between Different Analysis Types Test . . . 10
3.1.5 Error Message Validation Test . . . 11
3.1.6 Pagination and Data Loading Test . . . 11
3.1.7 Session Management Test . . . 12
3.1.8 Language and Special Character Handling Test . . . 12

4 Testing Findings **12**
4.1 White Box Testing Findings . . . 13
4.2 Black Box Testing Findings . . . 13

5 Recommendations **13**

6 Conclusion **14**

1 Introduction

Analytica is a web application designed to analyze Twitter data for sentiment, emotion, and toxicity. The application consists of a Django backend and a React frontend. This report documents the testing approach, test cases, and findings from both white box and black box testing perspectives.

1.1 Testing Objectives

- Evaluate the correctness of internal implementation (white box testing)
- Assess the functionality from an end-user perspective (black box testing)
- Identify potential bugs, performance issues, and areas for improvement
- Validate the application's core features performing unit, integration and system testing.

2 White Box Testing

White Box Testing, also known as Glass Box Testing, is a testing technique where the tester has full knowledge of the internal structure, code, and logic of the application.

2.1 Backend White Box Test Cases

2.1.1 Analyzer Classes Initialization Test

Test Case: Verify the singleton pattern in analyzer classes

```python
def test_analyzer_singleton_pattern():
    # Get two instances of the same analyzer
    analyzer1 = SentimentAnalyzer()
    analyzer2 = SentimentAnalyzer()

    # Test that they are the same instance
    assert analyzer1 is analyzer2

    # Same for other analyzer types
    emotion1 = EmotionAnalyzer()
    emotion2 = EmotionAnalyzer()
    assert emotion1 is emotion2

    toxicity1 = ToxicityAnalyzer()
    toxicity2 = ToxicityAnalyzer()
    assert toxicity1 is toxicity2
```

Objective: Validate that the singleton pattern is correctly implemented to ensure that only one instance of each analyzer is created.
Result: PASS - The singleton pattern is correctly implemented across all analyzer classes.

2.1.2 Language Detection and Translation Test

Test Case: Verify language detection and translation functions

```python
def test_language_detection_and_translation():
    # Test English detection
    english_text = "Hello, this is a test."
    detected_lang = detect_language(english_text)
    assert detected_lang == 'en'

    # Test non-English detection and translation
    spanish_text = "Hola, esto es una prueba."
    detected_lang = detect_language(spanish_text)
    assert detected_lang != 'en'

    translated = translate_to_english(spanish_text)
    assert translated != spanish_text
    assert detect_language(translated) == 'en'
```

Objective: Verify that the language detection function correctly identifies languages and that non-English text is properly translated to English.
Result: PASS with limitations - The functions work correctly for clear examples of different languages, but may have reduced accuracy with very short texts or mixed language content.

2.1.3 Tweet Analysis Pipeline Test

Test Case: Verify the entire analysis pipeline

```python
def test_tweet_analysis_pipeline():
    # Create a test tweet
    test_tweet = {
        'content': "I'm really happy about this product!",
        'tweet_id': '123456',
        'handle': '@testuser',
        'timestamp': '2024-03-31T12:00:00Z'
    }

    # Test sentiment analysis
    sentiment = analyze_tweet(test_tweet['content'], "sentiment")
    assert sentiment in ["Positive", "Negative", "Neutral"]

    # Test emotion analysis
    emotion = analyze_tweet(test_tweet['content'], "emotion")
    assert emotion in ["anger", "joy", "optimism", "sadness"]

    # Test toxicity analysis
    toxicity = analyze_tweet(test_tweet['content'], "toxicity")
    assert toxicity in ["offensive", "not-offensive"]
```

Objective: Validate that the tweet analysis pipeline works correctly for different types of analysis.
Result: PASS - The analysis pipeline produces expected results for all analysis types.

2.1.4 API Endpoint Security Test

Test Case: Verify authentication for history endpoints

```python
def test_history_api_authentication():
    # Test unauthenticated access
    client = Client()
    response = client.get('/api/history/')
    assert response.status_code == 401  # Should require authentication

    # Test with authentication
    client.login(username='testuser', password='testpassword')
    response = client.get('/api/history/')
    assert response.status_code == 200  # Should be accessible
```

Objective: Verify that history endpoints require user authentication.
Result: PASS - History endpoints correctly enforce authentication.

2.2 Frontend White Box Test Cases

2.2.1 React Component State Management Test

Test Case: Verify state management in Analysis component

```javascript
test('Analysis component updates state on form submission', async () => {
  // Mock the API response
  global.fetch = jest.fn().mockResolvedValue({
    ok: true,
    json: jest.fn().mockResolvedValue([{
      content: 'Test tweet',
      handle: '@test',
      timestamp: '2024-03-31T12:00:00Z',
      sentiment: 'Positive'
    }])
  });

  // Render the component
  const { getByText, getByLabelText } = render(<Analysis />);

  // Fill the form
  fireEvent.change(getByLabelText('Enter hashtag...'),
    { target: { value: '#test' } });

  // Submit form
  fireEvent.click(getByText('Analyze !'));

  // Wait for state update
  await waitFor(() => {
    expect(getByText('Test tweet')).toBeInTheDocument();
  });
});
```

Objective: Verify that React components correctly update their state in response to user actions.
Result: PASS - Component state is correctly updated after form submission.

2.2.2 Route Navigation and Authentication Test

Test Case: Verify route protection and redirection

```
test('Protected routes redirect unauthenticated users to login', () => {
  // Mock authentication state to be false
  jest.mock('../utils/auth', () => ({
    isAuthenticated: jest.fn(() => false)
  }));

  const { isAuthenticated } = require('../utils/auth');

  // Setup router with history
  const history = createMemoryHistory();
  history.push('/history');

  // Render the App component with router
  render(
    <Router history={history}>
      <App />
    </Router>
  );

  // Check if redirected to login page
  expect(history.location.pathname).toBe('/login');
  expect(isAuthenticated).toHaveBeenCalled();
});
```

Objective: Verify that protected routes redirect unauthenticated users to the login page.
Result: PASS - Unauthenticated users are correctly redirected to the login page when attempting to access protected routes.

2.2.3 API Error Handling Test

Test Case: Verify error handling in API requests

```
test('Analysis component handles API errors correctly', async () => {
  // Mock API error response
  global.fetch = jest.fn().mockResolvedValue({
    ok: false,
    status: 500,
    text: jest.fn().mockResolvedValue('Internal Server Error')
  });

  // Render the component
  const { getByText, findByText } = render(<Analysis />);

  // Submit form to trigger API call
  fireEvent.click(getByText('Analyze!'));

  // Check if error message is displayed
  const errorMessage = await findByText(/Analysis failed: 500/i);
  expect(errorMessage).toBeInTheDocument();

  // Verify error state is set correctly
  expect(screen.queryByTestId('loading-indicator')).not.toBeInTheDocument
    ();
  expect(screen.queryByTestId('results-container')).not.toBeInTheDocument
    ();
});
```

Objective: Verify that the application handles API errors gracefully and displays appropriate error messages.
Result: PARTIAL PASS - The application displays error messages but some error handling could be improved for more specific error information.

2.2.4 TweetCard Component Rendering Test

Test Case: Verify TweetCard component rendering with different analysis types

```
test('TweetCard renders correctly with different analysis types', () => {
  const tweetData = {
    content: 'This is a test tweet',
    handle: '@testuser',
    timestamp: '2024-03-31T12:00:00Z',
    tweet_id: '123456'
  };

  const analysis = {
    sentiment: 'Positive',
    emotion: 'joy',
    toxicity: 'not-offensive'
  };

  // Test rendering with sentiment analysis
  const { getByText, rerender } = render(
    <TweetCard
      tweet={tweetData}
      analysis={analysis}
      type="sentimental"
    />
  );

  // Check sentiment is displayed but not emotion or toxicity
  expect(getByText('Sentiment:')).toBeInTheDocument();
  expect(getByText('Positive')).toBeInTheDocument();
  expect(screen.queryByText('Emotion:')).not.toBeInTheDocument();
  expect(screen.queryByText('Toxicity:')).not.toBeInTheDocument();

  // Rerender with emotion analysis
  rerender(
    <TweetCard
      tweet={tweetData}
      analysis={analysis}
      type="emotional"
    />
  );

  // Check emotion is displayed but not sentiment or toxicity
  expect(getByText('Emotion:')).toBeInTheDocument();
  expect(getByText('joy')).toBeInTheDocument();
  expect(screen.queryByText('Sentiment:')).not.toBeInTheDocument();

  // Rerender with combined analysis
  rerender(
    <TweetCard
      tweet={tweetData}
      analysis={analysis}
      type="combined"
    />
  );

  // Check all analysis types are displayed
  expect(getByText('Sentiment:')).toBeInTheDocument();
  expect(getByText('Emotion:')).toBeInTheDocument();
  expect(getByText('Toxicity:')).toBeInTheDocument();
});
```

Objective: Verify that the TweetCard component correctly renders different types of analysis results based on the specified type.
Result: PASS - The TweetCard component correctly displays the appropriate analysis results based on the analysis type.

2.2.5 Authentication Flow Test

Test Case: Verify login form submission and authentication flow

```
test('Login component correctly handles authentication flow', async () =>
    {
  // Mock successful authentication response
  global.fetch = jest.fn().mockResolvedValue({
    ok: true,
    json: jest.fn().mockResolvedValue({
      message: 'Logged in successfully',
      user_id: 1,
      username: 'testuser'
    })
  });

  // Setup mock navigation
  const mockNavigate = jest.fn();
  jest.mock('react-router-dom', () => ({
    ...jest.requireActual('react-router-dom'),
    useNavigate: () => mockNavigate
  }));

  // Render login component
  const { getByLabelText, getByText } = render(<Login />);

  // Fill login form
  fireEvent.change(getByLabelText('Username'), {
    target: { value: 'testuser' }
  });
  fireEvent.change(getByLabelText('Password'), {
    target: { value: 'password123' }
  });

  // Submit form
  fireEvent.click(getByText('Login'));

  // Wait for API call to complete
  await waitFor(() => {
    // Verify navigation to home page
    expect(mockNavigate).toHaveBeenCalledWith('/');

    // Verify form was submitted with correct data
    expect(global.fetch).toHaveBeenCalledWith(
      'http://localhost:8000/api/auth/login/',
      expect.objectContaining({
        method: 'POST',
        body: JSON.stringify({
          username: 'testuser',
          password: 'password123'
        })
      })
    );
  });
});
```

Objective: Verify that the login component correctly handles the authentication flow, including form submission and redirection after successful login.
Result: PASS - The login component correctly handles form submission, API interaction, and navigation after successful login.

3 Black Box Testing

Black Box Testing, also known as Functional Testing, is a software testing method where the tester evaluates the software without knowledge of its internal code or structure.

3.1 Frontend Black Box Test Cases

3.1.1 User Registration Test

Test Case: User Registration

Objective: Verify that a new user can register successfully.
Steps:

1. Navigate to the registration page
2. Enter username, email, and password
3. Click the Register button
4. Verify redirection to login page

Expected Result: User account is created and user is redirected to login page.
Actual Result: PASS with observations - Registration works correctly but error messages for invalid inputs (e.g., username already taken) could be more descriptive.

3.1.2 Tweet Analysis with Different Parameters Test

Test Case: Tweet Analysis with Different Parameters

Objective: Verify that the system can analyze tweets using different parameters.
Steps:

1. Login to the application
2. Navigate to the Analysis page
3. Select 'Sentiment' analysis type
4. Enter hashtag '#technology'
5. Set number of tweets to 10
6. Click Analyze button
7. Verify results are displayed

Expected Result: System fetches and analyzes 10 tweets related to #technology, displaying the sentiment analysis results.
Actual Result: PASS with performance concerns - Analysis works as expected, but there's some delay when analyzing a large number of tweets (>50).

3.1.3 Search History Retrieval Test

Test Case: Search History Retrieval

Objective: Verify that user can access their search history.
Steps:

1. Login to the application
2. Navigate to the History page
3. Verify previous searches are listed
4. Click on a specific search
5. Verify that analysis results for that search are displayed

Expected Result: User can see all previous searches and can access the detailed results for each search.
Actual Result: PASS.

3.1.4 Navigation Between Different Analysis Types Test

Test Case: Navigation Between Analysis Types

Objective: Verify that user can switch between different analysis types.
Steps:

1. Login to the application
2. Navigate to Sentiment Analysis page
3. Perform an analysis
4. Navigate to Emotion Analysis page
5. Perform another analysis
6. Check that both analyses are saved in history

Expected Result: User can switch between different analysis types, and each analysis is saved separately in history.
Actual Result: PASS - Navigation between analysis types works as expected, and analyses are correctly saved in history.

3.1.5 Error Message Validation Test

Test Case: Error Message Validation

Objective: Verify that appropriate error messages are displayed for invalid inputs.
Steps:

1. Login to the application
2. Navigate to Analysis page
3. Enter an invalid hashtag (e.g., containing special characters like "#test!")
4. Submit the form
5. Observe error message
6. Enter a non-existent username
7. Submit the form
8. Observe error message

Expected Result: Clear, user-friendly error messages should be displayed for each error condition.
Actual Result: NEEDS IMPROVEMENT - Error messages are displayed but could be more specific. For example, when a username doesn't exist, it simply displays "No tweets found" rather than specifically indicating the username is invalid.

3.1.6 Pagination and Data Loading Test

Test Case: Pagination and Data Loading

Objective: Verify that the application handles large result sets appropriately.
Steps:

1. Login to the application
2. Navigate to Analysis page
3. Enter a popular hashtag (e.g., "#news")
4. Set number of tweets to maximum (100)
5. Submit for analysis
6. Observe loading behavior and result presentation

Expected Result: Application should display a loading indicator during analysis and then present results in a paginated format for easy navigation.
Actual Result: PARTIAL PASS - Loading indicator is displayed, but all results are loaded at once on a single page, causing scrolling issues for large result sets. Pagination is not implemented.

3.1.7 Session Management Test

Test Case: Session Management

Objective: Verify that user sessions are managed correctly.
Steps:

1. Login to the application
2. Close the browser without logging out
3. Reopen the browser and navigate to the application
4. Check if session is maintained
5. Navigate to a different site then return to application
6. Check if session is maintained
7. Remain inactive for 30 minutes
8. Attempt to perform an action

Expected Result: Session should persist when browser is reopened or after navigating away and returning. Session should expire after prolonged inactivity.
Actual Result: PASS - Sessions are handled correctly with appropriate timeout settings.

3.1.8 Language and Special Character Handling Test

Test Case: Language and Special Character Handling

Objective: Verify that tweets in different languages and with special characters are handled correctly.
Steps:

1. Login to the application
2. Navigate to Analysis page
3. Enter a hashtag known to contain multilingual content (e.g., "#tokyo2021")
4. Submit for analysis
5. Check if non-English tweets are displayed correctly
6. Check if emojis and special characters render properly

Expected Result: Application should correctly display tweets in various languages and properly render emojis and special characters.
Actual Result: PASS WITH LIMITATIONS - Tweets in major languages display correctly and are translated for analysis. Most emojis render correctly, but some specialized Unicode characters appear as placeholders.

4 Testing Findings

4.1 White Box Testing Findings

1. **Singleton Pattern Implementation:** The singleton pattern in analyzer classes works correctly, ensuring that only one instance of each analyzer is created.
2. **Language Detection:** The language detection function correctly identifies English and non-English texts. However, for very short texts or texts with mixed languages, detection accuracy is reduced.
3. **Translation Pipeline:** The translation function works for major languages but can produce inaccurate translations for uncommon languages or specific technical terminology.
4. **Error Handling:** The application has inconsistent error handling in different components. Some parts properly catch and report errors, while others might fail silently.
5. **Component Rendering:** The TweetCard component correctly displays different types of analysis results based on the specified type (sentiment, emotion, toxicity, or combined).
6. **Route Protection:** The application successfully redirects unauthenticated users when they attempt to access protected routes.

4.2 Black Box Testing Findings

1. **User Registration:** Registration works correctly for valid inputs, but error messages for invalid inputs (e.g., username already taken) could be more descriptive.
2. **Analysis Functionality:** The tweet analysis works for both hashtags and usernames. However, there's a significant delay when analyzing a large number of tweets (>50).
3. **History Feature:** The search history feature works correctly, allowing users to view and access their previous analyses.
4. **Navigation:** The navigation between different analysis types is intuitive, and analyses are correctly saved in history.
5. **Error Messages:** Error messages are displayed but lack specificity, making it difficult for users to understand and resolve issues.
6. **Large Result Sets:** The application displays a loading indicator during analysis, but all results are loaded at once without pagination, causing potential performance issues with large result sets.
7. **Session Management:** Session handling is implemented correctly with appropriate timeout settings and persistence across page navigations.
8. **Multilingual Support:** The application can handle and translate non-English tweets, though there are limitations with certain languages and specialized Unicode characters.

5 Recommendations

Based on the findings from both white box and black box testing, we recommend the following improvements:

1. **Improve Error Handling:** Implement consistent error handling across all components and provide more descriptive error messages to users, clearly indicating the specific issue and suggesting possible resolutions.

2. **Optimize Performance:** Optimize the tweet scraping and analysis process to reduce response times, particularly for large tweet sets, by implementing server-side caching and more efficient database queries.

3. **Add Pagination:** Implement pagination for search results and history to improve performance and usability when dealing with large data sets.

4. **Enhance Translation:** Improve translation quality for technical terms and specific domains by using domain-specific translation models or glossaries.

5. **Improve Error Messages:** Make error messages more specific and helpful to guide users in resolving issues, particularly for cases like non-existent usernames or invalid hashtags.

6. **Enhance Multilingual Support:** Improve handling of special characters and less common languages to provide more accurate analysis of non-English content.

6 Conclusion

The testing has validated that the Analytica application functions correctly for its core features, with both white box and black box testing approaches providing valuable insights. The application successfully performs sentiment, emotion, and toxicity analysis on tweets and provides a history feature for users to review past analyses.

The white box testing confirmed the proper implementation of key architectural patterns like the singleton pattern for analyzers and appropriate route protection mechanisms. Meanwhile, black box testing validated the user experience flow while identifying opportunities for improvement in areas like error messaging and handling large result sets.