

задание 1

```

import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.cluster import KMeans
from sklearn.preprocessing import StandardScaler
import warnings
warnings.filterwarnings('ignore')

class GameificationAnalyzer:
    """
    Расширенный класс для анализа эффективности геймификации в банковских приложениях
    """

    def __init__(self, user_data):
        self.user_data = user_data
        self.df = pd.DataFrame(user_data)
        self.engagement_scores = None
        self.roi_data = None

    def calculate_engagement_score(self, weights=None):
        """
        Расчет индекса вовлеченности пользователей с весовыми коэффициентами

        Args:
            weights (dict): Словарь с весами для различных метрик
        """
        if weights is None:
            weights = {
                'usage_frequency': 0.4,
                'time_spent': 0.3,
                'features_used': 0.2,
                'achievements_unlocked': 0.1
            }

        # Нормализация метрик
        max_frequency = self.df['usage_frequency'].max()
        max_time = self.df['time_spent'].max()
        max_features = self.df['features_used'].max()
        max_achievements = self.df.get('achievements_unlocked', pd.Series([1])).max()

        scores = []
        for _, user in self.df.iterrows():
            # Нормализованные компоненты
            freq_norm = (user['usage_frequency'] / max_frequency) * 100
            time_norm = (user['time_spent'] / max_time) * 100
            features_norm = (user['features_used'] / max_features) * 100
            achievements_norm = (user.get('achievements_unlocked', 0) / max_achievements) * 100

            # Взвешенная сумма
            score = (freq_norm * weights['usage_frequency'] +
                     time_norm * weights['time_spent'] +
                     features_norm * weights['features_used'] +
                     achievements_norm * weights['achievements_unlocked'])

            scores.append(score)

        self.engagement_scores = np.array(scores)
        self.df['engagement_score'] = self.engagement_scores
        return self.engagement_scores

    def segment_users(self, method='threshold'):
        """
        Сегментация пользователей по уровню вовлеченности

        Args:
            method (str): Метод сегментации ('threshold', 'clustering')
        """
        if self.engagement_scores is None:
            self.calculate_engagement_score()

        if method == 'threshold':
            # Сегментация по пороговым значениям
            conditions = [
                self.df['engagement_score'] >= 80,
                (self.df['engagement_score'] >= 40) & (self.df['engagement_score'] < 80),
                self.df['engagement_score'] < 40
            ]
    
```

```

choices = ['VIP', 'Active', 'Casual']
self.df['segment'] = np.select(conditions, choices, default='Unknown')

elif method == 'clustering':
    # Кластерный анализ для сегментации
    features = ['engagement_score', 'usage_frequency', 'time_spent', 'features_used']
    X = self.df[features]

    # Стандартизация данных
    scaler = StandardScaler()
    X_scaled = scaler.fit_transform(X)

    # K-means кластеризация
    kmeans = KMeans(n_clusters=3, random_state=42, n_init=10)
    clusters = kmeans.fit_predict(X_scaled)

    # Сопоставление кластеров с сегментами
    cluster_stats = self.df.groupby(clusters)['engagement_score'].mean()
    cluster_mapping = {}

    # Сортируем кластеры по среднему уровню вовлеченности
    sorted_clusters = cluster_stats.sort_values(ascending=False).index
    segment_names = ['VIP', 'Active', 'Casual']

    for i, cluster_id in enumerate(sorted_clusters):
        cluster_mapping[cluster_id] = segment_names[i]

    self.df['segment'] = [cluster_mapping[cluster] for cluster in clusters]

return self.df

def calculate_roi(self, gamification_costs, revenue_per_vip=1000, revenue_per_active=500, revenue_per_casual=100):
    """
    Расчет ROI геймификации

    Args:
        gamification_costs (float): Общие затраты на геймификацию
        revenue_per_* (float): Доход от разных сегментов пользователей
    """
    if 'segment' not in self.df.columns:
        self.segment_users()

    # Расчет дополнительного дохода от геймификации
    segment_revenue = {
        'VIP': revenue_per_vip,
        'Active': revenue_per_active,
        'Casual': revenue_per_casual
    }

    segment_counts = self.df['segment'].value_counts()
    total_revenue = 0

    for segment, count in segment_counts.items():
        total_revenue += count * segment_revenue.get(segment, 0)

    # Расчет ROI
    roi = ((total_revenue - gamification_costs) / gamification_costs) * 100

    self.roi_data = {
        'total_revenue': total_revenue,
        'gamification_costs': gamification_costs,
        'roi': roi,
        'segment_counts': segment_counts,
        'segment_revenue': segment_revenue
    }

    return roi

def analyze_behavioral_patterns(self):
    """
    Анализ поведенческих паттернов пользователей
    """
    behavioral_analysis = {}

    # Анализ по сегментам
    segment_analysis = self.df.groupby('segment').agg({
        'engagement_score': ['mean', 'std', 'count'],
        'usage_frequency': 'mean',
        'time_spent': 'mean',
        'features_used': 'mean'
    }).round(2)

```

```

behavioral_analytics[segment_stats] = segment_analysis

# Корреляционный анализ
numeric_columns = ['engagement_score', 'usage_frequency', 'time_spent', 'features_used']
correlation_matrix = self.df[numeric_columns].corr()
behavioral_analysis['correlation_matrix'] = correlation_matrix

# Анализ конверсии между сегментами
total_users = len(self.df)
segment_percentage = (self.df['segment'].value_counts() / total_users * 100).round(1)
behavioral_analysis['segment_distribution'] = segment_percentage

return behavioral_analysis

def generate_recommendations(self):
    """
    Генерация рекомендаций по улучшению геймификации
    """
    recommendations = []

    segment_dist = self.df['segment'].value_counts(normalize=True) * 100

    # Анализ Casual пользователей
    if segment_dist.get('Casual', 0) > 40:
        recommendations.append(
            "Высокий процент Casual пользователей. Рекомендуется внедрить систему " +
            "микро-наград за ежедневное использование приложения."
        )

    # Анализ VIP пользователей
    if segment_dist.get('VIP', 0) < 20:
        recommendations.append(
            "Низкий процент VIP пользователей. Рассмотрите внедрение эксклюзивных " +
            "премиум-функций и персонализированных предложений."
        )

    # Анализ вовлеченности
    avg_engagement = self.df['engagement_score'].mean()
    if avg_engagement < 60:
        recommendations.append(
            f"Средняя вовлеченность ({avg_engagement:.1f}) ниже оптимального уровня. " +
            "Рекомендуется добавить прогрессивную систему уровней и социальные элементы."
        )

    return recommendations

def visualize_analysis(self):
    """
    Визуализация результатов анализа
    """
    fig, axes = plt.subplots(2, 2, figsize=(15, 12))
    fig.suptitle('Анализ эффективности геймификации', fontsize=16, fontweight='bold')

    # 1. Распределение сегментов пользователей
    segment_counts = self.df['segment'].value_counts()
    colors = ['#ff9999', '#66b3ff', '#99ff99']
    axes[0, 0].pie(segment_counts.values, labels=segment_counts.index, autopct='%1.1f%%',
                  colors=colors[:len(segment_counts)])
    axes[0, 0].set_title('Распределение сегментов пользователей')

    # 2. Распределение индекса вовлеченности
    axes[0, 1].hist(self.df['engagement_score'], bins=20, alpha=0.7, color='skyblue', edgecolor='black')
    axes[0, 1].axvline(self.df['engagement_score'].mean(), color='red', linestyle='--',
                      label=f'Среднее: {self.df["engagement_score"].mean():.1f}')
    axes[0, 1].set_xlabel('Индекс вовлеченности')
    axes[0, 1].set_ylabel('Количество пользователей')
    axes[0, 1].set_title('Распределение индекса вовлеченности')
    axes[0, 1].legend()

    # 3. Метрики по сегментам
    segment_metrics = self.df.groupby('segment')[['usage_frequency', 'time_spent', 'features_used']].mean()
    segment_metrics.plot(kind='bar', ax=axes[1, 0], colormap='viridis')
    axes[1, 0].set_title('Средние метрики по сегментам')
    axes[1, 0].set_ylabel('Значение')
    axes[1, 0].tick_params(axis='x', rotation=45)

    # 4. ROI и экономические показатели
    if self.roi_data:
        metrics = ['VIP', 'Active', 'Casual']
        revenue_values = [self.roi_data['segment_counts'].get(seg, 0) *
                          self.roi_data['segment_revenue'][seg] for seg in metrics]

        axes[1, 1].bar(metrics, revenue_values, color=['gold', 'silver', 'brown'])

        axes[1, 1].bar(metrics, revenue_values, color=['gold', 'silver', 'brown'])

```

```

    axes[1, 1].set_title(f'Доход по сегментам (ROI: {self.roi_data["roi"]:.1f}%)')
    axes[1, 1].set_ylabel('Доход')

    plt.tight_layout()
    return fig

def generate_report(self, gamification_costs=50000):
    """
    Генерация полного отчета по анализу геймификации
    """

    # Выполняем все расчеты
    self.calculate_engagement_score()
    self.segment_users(method='threshold')
    roi = self.calculate_roi(gamification_costs)
    behavioral_patterns = self.analyze_behavioral_patterns()
    recommendations = self.generate_recommendations()

    # Формируем отчет
    report = {
        'overview': {
            'total_users': len(self.df),
            'average_engagement': round(self.df['engagement_score'].mean(), 1),
            'engagement_std': round(self.df['engagement_score'].std(), 1)
        },
        'segmentation': {
            'distribution': behavioral_patterns['segment_distribution'].to_dict(),
            'segment_stats': behavioral_patterns['segment_stats'].to_dict()
        },
        'roi_analysis': self.roi_data,
        'correlation_analysis': behavioral_patterns['correlation_matrix'].to_dict(),
        'recommendations': recommendations
    }

    return report

# Пример использования
if __name__ == "__main__":
    # Генерация тестовых данных
    np.random.seed(42)
    n_users = 1000

    user_data = []
    for i in range(n_users):
        user = {
            'user_id': f'user_{i+1:04d}',
            'usage_frequency': np.random.randint(1, 30),  # раз в месяц
            'time_spent': np.random.randint(5, 120),      # минут в неделю
            'features_used': np.random.randint(1, 15),     # количество функций
            'achievements_unlocked': np.random.randint(0, 10) # разблокированные достижения
        }
        user_data.append(user)

    # Создание анализатора и выполнение анализа
    analyzer = GameificationAnalyzer(user_data)

    # Генерация полного отчета
    report = analyzer.generate_report(gamification_costs=75000)

    # Вывод ключевых метрик
    print("=" * 50)
    print("ОТЧЕТ ПО АНАЛИЗУ ГЕЙМИФИКАЦИИ")
    print("=" * 50)
    print(f"Всего пользователей: {report['overview']['total_users']}")
    print(f"Средняя вовлеченность: {report['overview']['average_engagement']} баллов")
    print(f"ROI геймификации: {report['roi_analysis']['roi']:.1f}%")
    print("\nРаспределение сегментов:")
    for segment, percentage in report['segmentation']['distribution'].items():
        print(f" {segment}: {percentage}%")

    print("\nРекомендации:")
    for i, recommendation in enumerate(report['recommendations'], 1):
        print(f"{i}. {recommendation}")

    # Визуализация результатов
    fig = analyzer.visualize_analysis()
    plt.show()

```

```
=====
ОТЧЕТ ПО АНАЛИЗУ ГЕЙМИФИКАЦИИ
=====
Всего пользователей: 1000
Средняя вовлеченность: 51.0 баллов
ROI геймификации: 455.3%
```

Распределение сегментов:

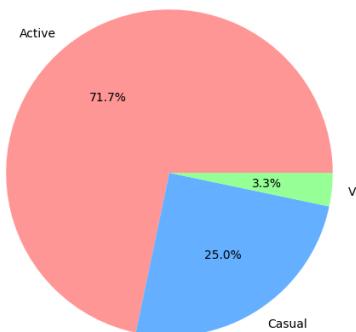
Active: 71.7%
 Casual: 25.0%
 VIP: 3.3%

Рекомендации:

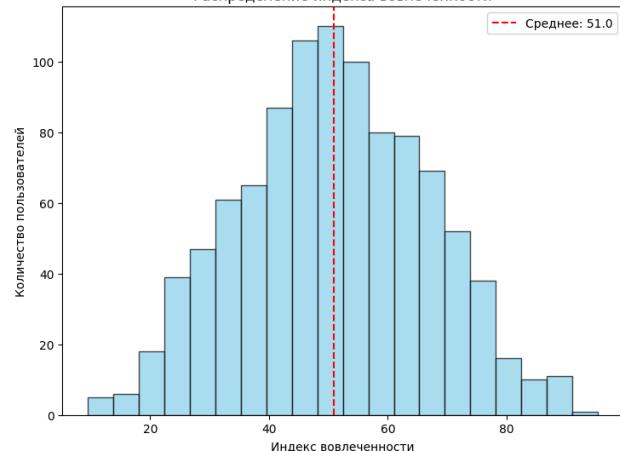
- Низкий процент VIP пользователей. Рассмотрите внедрение эксклюзивных премиум-функций и персонализированных предложений.
- Средняя вовлеченность (51.0) ниже оптимального уровня. Рекомендуется добавить прогрессивную систему уровней и социальные

Анализ эффективности геймификации

Распределение сегментов пользователей



Распределение индекса вовлеченности



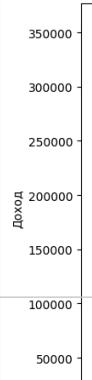
Средние метрики по сегментам

задание2

- usage_frequency
- time_spent
- features_used



Доход по сегментам (ROI: 455.3%)



```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.cluster import KMeans
from sklearn.preprocessing import StandardScaler
import warnings
warnings.filterwarnings('ignore')

class GameificationAnalyzer:
    """
    Расширенный класс для анализа эффективности геймификации в банковских приложениях
    """

    def __init__(self, user_data):
        self.user_data = user_data
        self.df = pd.DataFrame(user_data)
        self.engagement_scores = None
        self.roi_data = None

    def calculate_engagement_score(self, weights=None):
        """
        Расчет индекса вовлеченности пользователей с весовыми коэффициентами

        Args:
            weights (dict): Словарь с весами для различных метрик
        """
        if weights is None:
            weights = {
                'usage_frequency': 0.4,
                'time_spent': 0.3,
                'features_used': 0.2,
            }
        else:
            pass
```

```

        'achievements_unlocked': 0.1
    }

# Проверяем наличие колонки achievements_unlocked
if 'achievements_unlocked' not in self.df.columns:
    self.df['achievements_unlocked'] = 0

# Нормализация метрик
max_frequency = self.df['usage_frequency'].max() or 1
max_time = self.df['time_spent'].max() or 1
max_features = self.df['features_used'].max() or 1
max_achievements = self.df['achievements_unlocked'].max() or 1

scores = []
for _, user in self.df.iterrows():
    # Нормализованные компоненты
    freq_norm = (user['usage_frequency'] / max_frequency) * 100
    time_norm = (user['time_spent'] / max_time) * 100
    features_norm = (user['features_used'] / max_features) * 100
    achievements_norm = (user['achievements_unlocked'] / max_achievements) * 100

    # Взвешенная сумма
    score = (freq_norm * weights['usage_frequency'] +
              time_norm * weights['time_spent'] +
              features_norm * weights['features_used'] +
              achievements_norm * weights['achievements_unlocked'])

    scores.append(score)

self.engagement_scores = np.array(scores)
self.df['engagement_score'] = self.engagement_scores
return self.engagement_scores

def segment_users(self, method='threshold'):
    """
    Сегментация пользователей по уровню вовлеченности

    Args:
        method (str): Метод сегментации ('threshold', 'clustering')
    """

    if self.engagement_scores is None:
        self.calculate_engagement_score()

    if method == 'threshold':
        # Сегментация по пороговым значениям
        conditions = [
            self.df['engagement_score'] >= 80,
            (self.df['engagement_score'] >= 40) & (self.df['engagement_score'] < 80),
            self.df['engagement_score'] < 40
        ]
        choices = ['VIP', 'Active', 'Casual']
        self.df['segment'] = np.select(conditions, choices, default='Unknown')

    elif method == 'clustering':
        # Кластерный анализ для сегментации
        features = ['engagement_score', 'usage_frequency', 'time_spent', 'features_used']
        X = self.df[features]

        # Стандартизация данных
        scaler = StandardScaler()
        X_scaled = scaler.fit_transform(X)

        # K-means кластеризация
        kmeans = KMeans(n_clusters=3, random_state=42, n_init=10)
        clusters = kmeans.fit_predict(X_scaled)

        # Сопоставление кластеров с сегментами
        cluster_stats = self.df.groupby(clusters)['engagement_score'].mean()
        cluster_mapping = {}

        # Сортируем кластеры по среднему уровню вовлеченности
        sorted_clusters = cluster_stats.sort_values(ascending=False).index
        segment_names = ['VIP', 'Active', 'Casual']

        for i, cluster_id in enumerate(sorted_clusters):
            cluster_mapping[cluster_id] = segment_names[i]

        self.df['segment'] = [cluster_mapping.get(cluster, 'Unknown') for cluster in clusters]

    return self.df

def calculate_roi(self, gamification_costs, revenue_per_vip=1000, revenue_per_active=500, revenue_per_casual=100):

```

```

"""
Расчет ROI геймификации

Args:
    gamification_costs (float): Общие затраты на геймификацию
    revenue_per_* (float): Доход от разных сегментов пользователей
"""

if 'segment' not in self.df.columns:
    self.segment_users()

# Расчет дополнительного дохода от геймификации
segment_revenue = {
    'VIP': revenue_per_vip,
    'Active': revenue_per_active,
    'Casual': revenue_per_casual
}

segment_counts = self.df['segment'].value_counts()
total_revenue = 0

for segment, count in segment_counts.items():
    total_revenue += count * segment_revenue.get(segment, 0)

# Расчет ROI
roi = ((total_revenue - gamification_costs) / gamification_costs) * 100 if gamification_costs != 0 else 0

self.roi_data = {
    'total_revenue': total_revenue,
    'gamification_costs': gamification_costs,
    'roi': roi,
    'segment_counts': segment_counts.to_dict(),
    'segment_revenue': segment_revenue
}

return roi

def analyze_behavioral_patterns(self):
    """
    Анализ поведенческих паттернов пользователей
    """

    behavioral_analysis = {}

    # Анализ по сегментам
    segment_analysis = self.df.groupby('segment').agg({
        'engagement_score': ['mean', 'std', 'count'],
        'usage_frequency': 'mean',
        'time_spent': 'mean',
        'features_used': 'mean'
    }).round(2)

    behavioral_analysis['segment_stats'] = segment_analysis

    # Корреляционный анализ
    numeric_columns = ['engagement_score', 'usage_frequency', 'time_spent', 'features_used']
    correlation_matrix = self.df[numeric_columns].corr()
    behavioral_analysis['correlation_matrix'] = correlation_matrix

    # Анализ конверсии между сегментами
    total_users = len(self.df)
    segment_percentage = (self.df['segment'].value_counts() / total_users * 100).round(1)
    behavioral_analysis['segment_distribution'] = segment_percentage

    return behavioral_analysis

def generate_recommendations(self):
    """
    Генерация рекомендаций по улучшению геймификации
    """

    recommendations = []

    if 'segment' not in self.df.columns:
        self.segment_users()

    segment_dist = self.df['segment'].value_counts(normalize=True) * 100

    # Анализ Casual пользователей
    if segment_dist.get('Casual', 0) > 40:
        recommendations.append(
            "Высокий процент Casual пользователей. Рекомендуется внедрить систему " +
            "микро-наград за ежедневное использование приложения."
        )

```

```

# Анализ VIP пользователей
if segment_dist.get('VIP', 0) < 20:
    recommendations.append(
        "Низкий процент VIP пользователей. Рассмотрите внедрение эксклюзивных " +
        "премиум-функций и персонализированных предложений."
    )

# Анализ вовлеченности
avg_engagement = self.df['engagement_score'].mean()
if avg_engagement < 60:
    recommendations.append(
        f"Средняя вовлеченность ({avg_engagement:.1f}) ниже оптимального уровня. " +
        "Рекомендуется добавить прогрессивную систему уровней и социальные элементы."
    )

return recommendations

def visualize_analysis(self):
    """
    Визуализация результатов анализа
    """

    fig, axes = plt.subplots(2, 2, figsize=(15, 12))
    fig.suptitle('Анализ эффективности геймификации', fontsize=16, fontweight='bold')

    # 1. Распределение сегментов пользователей
    segment_counts = self.df['segment'].value_counts()
    colors = ['#ff9999', '#66b3ff', '#99ff99']
    axes[0, 0].pie(segment_counts.values, labels=segment_counts.index, autopct='%1.1f%%',
                  colors=colors[:len(segment_counts)])
    axes[0, 0].set_title('Распределение сегментов пользователей')

    # 2. Распределение индекса вовлеченности
    axes[0, 1].hist(self.df['engagement_score'], bins=20, alpha=0.7, color='skyblue', edgecolor='black')
    axes[0, 1].axvline(self.df['engagement_score'].mean(), color='red', linestyle='--',
                      label=f'Среднее: {self.df["engagement_score"].mean():.1f}')
    axes[0, 1].set_xlabel('Индекс вовлеченности')
    axes[0, 1].set_ylabel('Количество пользователей')
    axes[0, 1].set_title('Распределение индекса вовлеченности')
    axes[0, 1].legend()

    # 3. Метрики по сегментам
    segment_metrics = self.df.groupby('segment')[['usage_frequency', 'time_spent', 'features_used']].mean()
    segment_metrics.plot(kind='bar', ax=axes[1, 0], colormap='viridis')
    axes[1, 0].set_title('Средние метрики по сегментам')
    axes[1, 0].set_ylabel('Значение')
    axes[1, 0].tick_params(axis='x', rotation=45)

    # 4. ROI и экономические показатели
    if self.roi_data:
        metrics = ['VIP', 'Active', 'Casual']
        revenue_values = []
        for seg in metrics:
            count = self.roi_data['segment_counts'].get(seg, 0)
            revenue_per_user = self.roi_data['segment_revenue'].get(seg, 0)
            revenue_values.append(count * revenue_per_user)

        axes[1, 1].bar(metrics, revenue_values, color=['gold', 'silver', 'brown'])
        axes[1, 1].set_title(f'Доход по сегментам (ROI: {self.roi_data["roi"]:.1f}%)')
        axes[1, 1].set_ylabel('Доход')

    plt.tight_layout()
    return fig

def generate_report(self, gamification_costs=50000):
    """
    Генерация полного отчета по анализу геймификации
    """

    # Выполняем все расчеты
    self.calculate_engagement_score()
    self.segment_users(method='threshold')
    roi = self.calculate_roi(gamification_costs)
    behavioral_patterns = self.analyze_behavioral_patterns()
    recommendations = self.generate_recommendations()

    # Формируем отчет
    report = {
        'overview': {
            'total_users': len(self.df),
            'average_engagement': round(self.df['engagement_score'].mean(), 1),
            'engagement_std': round(self.df['engagement_score'].std(), 1)
        },
        'segmentation': {

```

```

'distribution': behavioral_patterns['segment_distribution'].to_dict(),
'segment_stats': behavioral_patterns['segment_stats'].to_dict()
},
'roi_analysis': self.roi_data,
'correlation_analysis': behavioral_patterns['correlation_matrix'].to_dict(),
'recommendations': recommendations
}

return report

# Пример использования с исправленными данными
if __name__ == "__main__":
    # Генерация тестовых данных
    np.random.seed(42)
    n_users = 1000

    user_data = []
    for i in range(n_users):
        user = {
            'user_id': f'user_{i+1:04d}',
            'usage_frequency': np.random.randint(1, 30), # раз в месяц
            'time_spent': np.random.randint(5, 120), # минут в неделю
            'features_used': np.random.randint(1, 15), # количество функций
            'achievements_unlocked': np.random.randint(0, 10) # разблокированные достижения
        }
        user_data.append(user)

    # Создание анализатора и выполнение анализа
    analyzer = GameificationAnalyzer(user_data)

    # Генерация полного отчета
    report = analyzer.generate_report(gamification_costs=75000)

    # Вывод ключевых метрик
    print("=" * 50)
    print("ОТЧЕТ ПО АНАЛИЗУ ГЕЙМИФИКАЦИИ")
    print("=" * 50)
    print(f"Всего пользователей: {report['overview']['total_users']} ")
    print(f"Средняя вовлеченность: {report['overview']['average_engagement']} баллов")
    print(f"ROI геймификации: {report['roi_analysis']['roi']:.1f}%")
    print("\nРаспределение сегментов:")
    for segment, percentage in report['segmentation']['distribution'].items():
        print(f"  {segment}: {percentage}%")

    print("\nРекомендации:")
    for i, recommendation in enumerate(report['recommendations'], 1):
        print(f"{i}. {recommendation}")

    # Визуализация результатов
    fig = analyzer.visualize_analysis()
    plt.show()

```

```
=====
ОТЧЕТ ПО АНАЛИЗУ ГЕЙМИФИКАЦИИ
=====
Всего пользователей: 1000
Средняя вовлеченность: 51.0 баллов
ROI геймификации: 455.3%
```

Распределение сегментов:

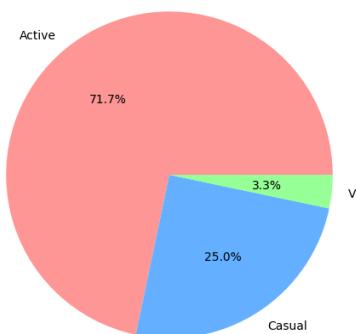
Active: 71.7%
 Casual: 25.0%
 VIP: 3.3%

Рекомендации:

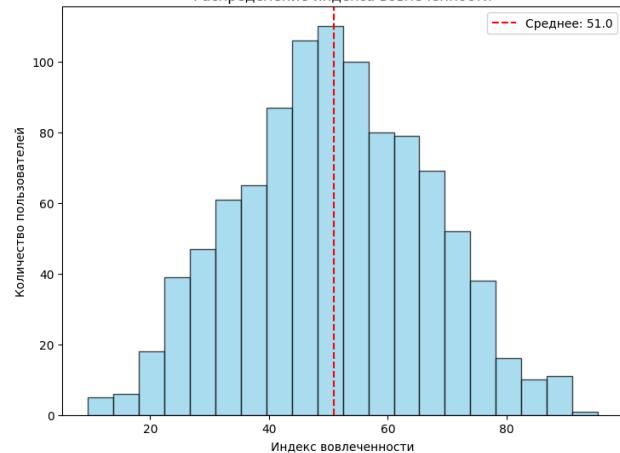
- Низкий процент VIP пользователей. Рассмотрите внедрение эксклюзивных премиум-функций и персонализированных предложений.
- Средняя вовлеченность (51.0) ниже оптимального уровня. Рекомендуется добавить прогрессивную систему уровней и социальные

Анализ эффективности геймификации

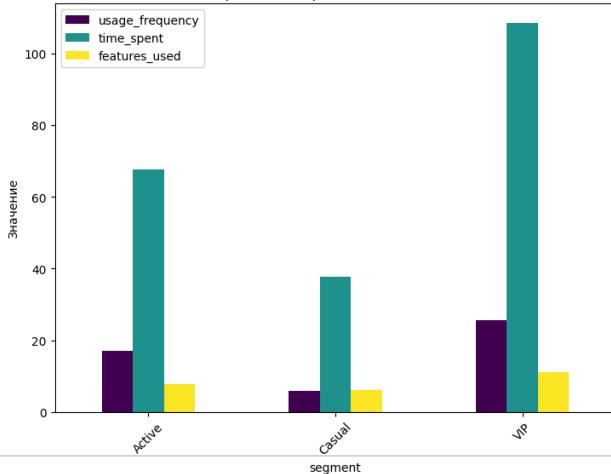
Распределение сегментов пользователей



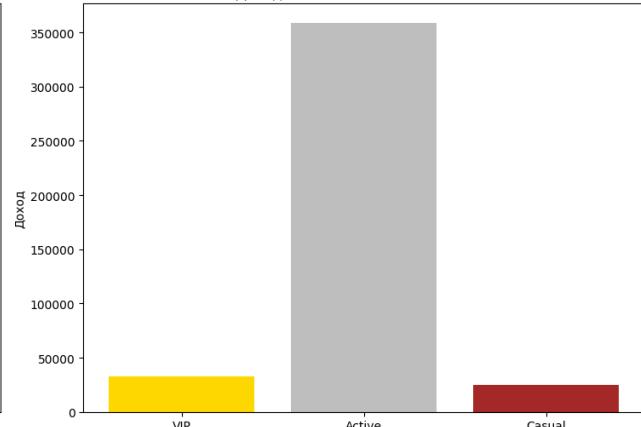
Распределение индекса вовлеченности



Средние метрики по сегментам



Доход по сегментам (ROI: 455.3%)



задание 3

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from datetime import datetime, timedelta
import warnings
warnings.filterwarnings('ignore')

class BigDataCustomerSegmentation:
    """
    Расширенный класс для RFM-анализа клиентов на основе больших данных
    """

    def __init__(self, transaction_data):
        """
        Инициализация с транзакционными данными

        Args:
            transaction_data (DataFrame): Данные транзакций
        """
        self.transaction_data = transaction_data
        self.current_date = transaction_data['date'].max() if 'date' in transaction_data.columns else datetime.now()
        self.rfm_data = None
        self.segmented_data = None

    def calculate_rfm_metrics(self):
        pass
```

```

"""
Расчет RFM метрик (Recency, Frequency, Monetary)
"""

# Преобразуем дату если необходимо
if not pd.api.types.is_datetime64_any_dtype(self.transaction_data['date']):
    self.transaction_data['date'] = pd.to_datetime(self.transaction_data['date'])

# Расчет RFM метрик
rfm_data = self.transaction_data.groupby('customer_id').agg({
    'date': lambda x: (self.current_date - x.max()).days, # Recency
    'transaction_id': 'count', # Frequency
    'amount': 'sum' # Monetary
}).round(2)

rfm_data.columns = ['Recency', 'Frequency', 'Monetary']
self.rfm_data = rfm_data

# Дополнительные метрики
self.rfm_data['Avg_Transaction_Value'] = self.rfm_data['Monetary'] / self.rfm_data['Frequency']
self.rfm_data['CLV'] = self.rfm_data['Monetary'] * self.rfm_data['Frequency'] / len(self.rfm_data)

return self.rfm_data

def calculate_rfm_scores(self, method='quantile'):
    """
    Расчет RFM scores (1-5 баллов для каждой метрики)

    Args:
        method (str): Метод расчета scores ('quantile', 'custom')
    """

    if self.rfm_data is None:
        self.calculate_rfm_metrics()

    if method == 'quantile':
        # Используем квантили для сегментации
        self.rfm_data['R_Score'] = pd.qcut(self.rfm_data['Recency'], 5, labels=[5, 4, 3, 2, 1])
        self.rfm_data['F_Score'] = pd.qcut(self.rfm_data['Frequency'], 5, labels=[1, 2, 3, 4, 5])
        self.rfm_data['M_Score'] = pd.qcut(self.rfm_data['Monetary'], 5, labels=[1, 2, 3, 4, 5])

    elif method == 'custom':
        # Кастомные границы для сегментации
        recency_bins = [0, 30, 90, 180, 365, float('inf')]
        frequency_bins = [0, 2, 5, 10, 20, float('inf')]
        monetary_bins = [0, 1000, 5000, 15000, 50000, float('inf')]

        self.rfm_data['R_Score'] = pd.cut(self.rfm_data['Recency'], bins=recency_bins, labels=[5, 4, 3, 2, 1])
        self.rfm_data['F_Score'] = pd.cut(self.rfm_data['Frequency'], bins=frequency_bins, labels=[1, 2, 3, 4, 5])
        self.rfm_data['M_Score'] = pd.cut(self.rfm_data['Monetary'], bins=monetary_bins, labels=[1, 2, 3, 4, 5])

    # Преобразуем в числовой формат
    self.rfm_data['R_Score'] = self.rfm_data['R_Score'].astype(int)
    self.rfm_data['F_Score'] = self.rfm_data['F_Score'].astype(int)
    self.rfm_data['M_Score'] = self.rfm_data['M_Score'].astype(int)

    # Комбинированный RFM score
    self.rfm_data['RFM_Score'] = (
        self.rfm_data['R_Score'].astype(str) +
        self.rfm_data['F_Score'].astype(str) +
        self.rfm_data['M_Score'].astype(str)
    ).astype(int)

    return self.rfm_data

def segment_customers(self):
    """
    Сегментация клиентов на основе RFM scores
    """

    if 'R_Score' not in self.rfm_data.columns:
        self.calculate_rfm_scores()

    # Создаем сегменты на основе RFM scores
    conditions = [
        (self.rfm_data['R_Score'] >= 4) & (self.rfm_data['F_Score'] >= 4) & (self.rfm_data['M_Score'] >= 4), # Champion
        (self.rfm_data['R_Score'] >= 4) & (self.rfm_data['F_Score'] >= 3) & (self.rfm_data['M_Score'] >= 3), # Loyal
        (self.rfm_data['R_Score'] >= 3) & (self.rfm_data['F_Score'] >= 3) & (self.rfm_data['M_Score'] >= 3), # Potent
        (self.rfm_data['R_Score'] >= 4) & (self.rfm_data['F_Score'] <= 2) & (self.rfm_data['M_Score'] <= 2), # New Customer
        (self.rfm_data['R_Score'] >= 3) & (self.rfm_data['F_Score'] <= 2) & (self.rfm_data['M_Score'] <= 2), # Promising
        (self.rfm_data['R_Score'] >= 2) & (self.rfm_data['F_Score'] <= 3) & (self.rfm_data['M_Score'] <= 3), # Need Attention
        (self.rfm_data['R_Score'] <= 2) & (self.rfm_data['F_Score'] >= 4) & (self.rfm_data['M_Score'] >= 4), # At Risk
        (self.rfm_data['R_Score'] <= 2) & (self.rfm_data['F_Score'] >= 3) & (self.rfm_data['M_Score'] >= 3), # Cannot Buy
        (self.rfm_data['R_Score'] <= 2) & (self.rfm_data['F_Score'] <= 2) & (self.rfm_data['M_Score'] <= 2), # Hibernating
        (self.rfm_data['R_Score'] <= 1) & (self.rfm_data['F_Score'] <= 1) & (self.rfm_data['M_Score'] <= 1) # Lost
    ]

```

```

    ]

    choices = [
        'Champions',
        'Loyal Customers',
        'Potential Loyalists',
        'New Customers',
        'Promising',
        'Need Attention',
        'At Risk',
        'Cannot Lose Them',
        'Hibernating',
        'Lost'
    ]

    self.rfm_data['Segment'] = np.select(conditions, choices, default='Other')
    self.segmented_data = self.rfm_data

    return self.rfm_data

def calculate_business_metrics(self):
    """
    Расчет ключевых бизнес-метрик
    """
    if self.segmented_data is None:
        self.segment_customers()

    total_revenue = self.rfm_data['Monetary'].sum()
    total_customers = len(self.rfm_data)
    avg_clv = self.rfm_data['CLV'].mean()

    segment_stats = self.rfm_data.groupby('Segment').agg({
        'Monetary': ['count', 'sum', 'mean'],
        'Frequency': 'mean',
        'Recency': 'mean',
        'CLV': 'mean'
    }).round(2)

    segment_stats.columns = ['Count', 'Total_Revenue', 'Avg_Revenue', 'Avg_Frequency', 'Avg_Recency', 'Avg_CLV']

    business_metrics = {
        'total_revenue': total_revenue,
        'total_customers': total_customers,
        'avg_clv': avg_clv,
        'segment_distribution': self.rfm_data['Segment'].value_counts().to_dict(),
        'segment_stats': segment_stats,
        'top_segment': segment_stats.nlargest(1, 'Total_Revenue').index[0]
    }

    return business_metrics

def generate_segment_recommendations(self):
    """
    Генерация рекомендаций для каждого сегмента
    """
    recommendations = {
        'Champions': [
            "Предлагать эксклюзивные премиум-услуги",
            "Внедрить программу лояльности с повышенными бонусами",
            "Персонализированные предложения на основе истории покупок"
        ],
        'Loyal Customers': [
            "Увеличить частоту коммуникации",
            "Предлагать кросс-SELLING продуктов",
            "Создать программу рекомендаций друзьям"
        ],
        'Potential Loyalists': [
            "Активные программы вовлечения",
            "Специальные предложения для увеличения частоты покупок",
            "Персональные консультации"
        ],
        'New Customers': [
            "Программа onboarding",
            "Специальные предложения для новых клиентов",
            "Образовательный контент о продуктах"
        ],
        'At Risk': [
            "Программы удержания со специальными условиями",
            "Персональные предложения от менеджера",
            "Опросы для выявления причин ухода"
        ],
        'Cannot Lose Them': [
    }

```

```

    "Агрессивные программы удержания",
    "Персональный менеджер",
    "Специальные условия и скидки"
]
}

return recommendations

def visualize_segmentation(self):
    """
    Визуализация результатов RFM-анализа
    """
    if self.segmented_data is None:
        self.segment_customers()

    fig, axes = plt.subplots(2, 3, figsize=(18, 12))
    fig.suptitle('RFM Анализ клиентских сегментов', fontsize=16, fontweight='bold')

    # 1. Распределение сегментов
    segment_dist = self.rfm_data['Segment'].value_counts()
    colors = plt.cm.Set3(np.linspace(0, 1, len(segment_dist)))
    axes[0, 0].pie(segment_dist.values, labels=segment_dist.index, autopct='%.1f%%',
                   colors=colors, startangle=90)
    axes[0, 0].set_title('Распределение клиентов по сегментам')

    # 2. Распределение Revenue по сегментам
    segment_revenue = self.rfm_data.groupby('Segment')['Monetary'].sum().sort_values(ascending=False)
    axes[0, 1].bar(segment_revenue.index, segment_revenue.values, color='lightcoral')
    axes[0, 1].set_title('Общий доход по сегментам')
    axes[0, 1].tick_params(axis='x', rotation=45)

    # 3. Средний CLV по сегментам
    segment_clv = self.rfm_data.groupby('Segment')['CLV'].mean().sort_values(ascending=False)
    axes[0, 2].bar(segment_clv.index, segment_clv.values, color='lightgreen')
    axes[0, 2].set_title('Средний CLV по сегментам')
    axes[0, 2].tick_params(axis='x', rotation=45)

    # 4. RFM распределение
    axes[1, 0].scatter(self.rfm_data['Recency'], self.rfm_data['Frequency'],
                       c=self.rfm_data['M_Score'], cmap='viridis', alpha=0.6)
    axes[1, 0].set_xlabel('Recency (дни)')
    axes[1, 0].set_ylabel('Frequency')
    axes[1, 0].set_title('RFM распределение (цвет = Monetary)')
    axes[1, 0].grid(True, alpha=0.3)

    # 5. Распределение Recency
    axes[1, 1].hist(self.rfm_data['Recency'], bins=30, alpha=0.7, color='skyblue', edgecolor='black')
    axes[1, 1].axvline(self.rfm_data['Recency'].mean(), color='red', linestyle='--',
                      label=f'Среднее: {self.rfm_data["Recency"].mean():.1f}')
    axes[1, 1].set_xlabel('Recency (дни)')
    axes[1, 1].set_ylabel('Количество клиентов')
    axes[1, 1].set_title('Распределение Recency')
    axes[1, 1].legend()

    # 6. Распределение Monetary
    axes[1, 2].hist(self.rfm_data['Monetary'], bins=30, alpha=0.7, color='gold', edgecolor='black')
    axes[1, 2].axvline(self.rfm_data['Monetary'].mean(), color='red', linestyle='--',
                      label=f'Среднее: {self.rfm_data["Monetary"].mean():.1f}')
    axes[1, 2].set_xlabel('Monetary (руб.)')
    axes[1, 2].set_ylabel('Количество клиентов')
    axes[1, 2].set_title('Распределение Monetary')
    axes[1, 2].legend()

plt.tight_layout()
return fig

def generate_detailed_report(self):
    """
    Генерация детального отчета по RFM-анализу
    """
    # Выполняем все расчеты
    self.calculate_rfm_metrics()
    self.calculate_rfm_scores()
    self.segment_customers()
    business_metrics = self.calculate_business_metrics()
    recommendations = self.generate_segment_recommendations()

    # Формируем отчет
    report = {
        'overview': {
            'total_customers': business_metrics['total_customers'],
            'total_revenue': business_metrics['total_revenue'],
            ...
        }
    }

    return report

```

```

        'avg_clv': business_metrics['avg_clv'],
        'top_segment': business_metrics['top_segment']
    },
    'segment_analysis': {
        'distribution': business_metrics['segment_distribution'],
        'statistics': business_metrics['segment_stats'].to_dict(),
        'top_segments_by_revenue': business_metrics['segment_stats'].nlargest(3, 'Total_Revenue')[['Total_Revenue'],
    },
    'rfm_metrics': {
        'recency_stats': self.rfm_data['Recency'].describe().to_dict(),
        'frequency_stats': self.rfm_data['Frequency'].describe().to_dict(),
        'monetary_stats': self.rfm_data['Monetary'].describe().to_dict()
    },
    'recommendations': recommendations
}

return report

```

Пример использования

```

def generate_sample_data(n_customers=500, n_transactions=10000):
    """
    Генерация тестовых данных для демонстрации
    """
    np.random.seed(42)

    customer_ids = [f'CUST_{i:04d}' for i in range(1, n_customers + 1)]

    transactions = []
    for i in range(n_transactions):
        customer_id = np.random.choice(customer_ids)
        amount = np.random.lognormal(8, 1.5) # Логнормальное распределение для сумм
        date = datetime.now() - timedelta(days=np.random.randint(1, 365))

        transaction = {
            'transaction_id': f'TXN_{i:06d}',
            'customer_id': customer_id,
            'amount': round(amount, 2),
            'date': date
        }
        transactions.append(transaction)

    return pd.DataFrame(transactions)

```

if __name__ == "__main__":
 # Генерация тестовых данных
 print("Генерация тестовых данных...")
 sample_data = generate_sample_data()

 # Создание анализатора
 analyzer = BigDataCustomerSegmentation(sample_data)

 # Генерация отчета
 report = analyzer.generate_detailed_report()

 # Вывод результатов
 print("=" * 60)
 print("ОТЧЕТ ПО RFM-АНАЛИЗУ КЛИЕНТОВ")
 print("=" * 60)
 print(f"Общий объем транзакций: {report['overview']['total_revenue']:.0f} руб.")
 print(f"Количество клиентов: {report['overview']['total_customers']} ")
 print(f"Средний CLV: {report['overview']['avg_clv']:.0f} руб.")
 print(f"Лучший сегмент: {report['overview']['top_segment']}")

 print("\nТоп-3 сегмента по доходу:")
 for segment, revenue in report['segment_analysis']['top_segments_by_revenue']['Total_Revenue'].items():
 count = report['segment_analysis']['top_segments_by_revenue'][Count][segment]
 print(f" {segment}: {revenue:.0f} руб. ({count} клиентов)")

 print("\nРаспределение сегментов:")
 for segment, count in report['segment_analysis']['distribution'].items():
 percentage = (count / report['overview']['total_customers']) * 100
 print(f" {segment}: {count} клиентов ({percentage:.1f}%)")

 print("\nКлючевые рекомендации:")
 for segment, segment_recommendations in report['recommendations'].items():
 if segment in report['segment_analysis']['distribution']:
 print(f"\n{segment}:")
 for i, recommendation in enumerate(segment_recommendations[:2], 1):
 print(f" {i}. {recommendation}")

Визуализация

```
fig = analyzer.visualize_segmentation()
plt.show()
```

Генерация тестовых данных...
=====
ОТЧЕТ ПО RFM-АНАЛИЗУ КЛИЕНТОВ
=====
Общий объем транзакций: 91,780,719 руб.
Количество клиентов: 500
Средний CLV: 7,675 руб.
Лучший сегмент: Other

Топ-3 сегмента по доходу:
Other: 25,050,844 руб. (118 клиентов)
Champions: 15,792,289 руб. (55 клиентов)
Need Attention: 11,234,176 руб. (93 клиентов)

Распределение сегментов:
Other: 118 клиентов (23.6%)
Need Attention: 93 клиентов (18.6%)
Champions: 55 клиентов (11.0%)
New Customers: 47 клиентов (9.4%)
Loyal Customers: 39 клиентов (7.8%)
Potential Loyalists: 35 клиентов (7.0%)
Hibernating: 34 клиентов (6.8%)
At Risk: 32 клиентов (6.4%)
Cannot Lose Them: 28 клиентов (5.6%)
Promising: 19 клиентов (3.8%)

Ключевые рекомендации:

Champions:

1. Предлагать эксклюзивные премиум-услуги
2. Внедрить программу лояльности с повышенными бонусами

Loyal Customers:

1. Увеличить частоту коммуникации
2. Предлагать кросс-SELLING продуктов

Potential Loyalists:

1. Активные программы вовлечения
2. Специальные предложения для увеличения частоты покупок

New Customers:

1. Программа onboarding
2. Специальные предложения для новых клиентов

At Risk:

1. Программы удержания со специальными условиями
2. Персональные предложения от менеджера

Cannot Lose Them:

1. Агрессивные программы удержания
2. Персональный менеджер

задание 4



```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.ensemble import RandomForestRegressor, GradientBoostingRegressor
from sklearn.linear_model import LinearRegression, Ridge, Lasso
from sklearn.svm import SVR
from sklearn.neural_network import MLPRegressor
from sklearn.metrics import mean_absolute_error, mean_squared_error, r2_score
from sklearn.model_selection import cross_val_score, GridSearchCV, TimeSeriesSplit
from sklearn.preprocessing import StandardScaler, RobustScaler
from sklearn.feature_selection import SelectKBest, f_regression
import warnings
warnings.filterwarnings('ignore')

class FinancialTrendPredictor:
    """
    Расширенный класс для прогнозирования финансовых трендов финтех-компаний
    """

    def __init__(self, random_state=42):
        self.random_state = random_state
        self.models = {}
        self.results = {}

    def fit(self, X, y):
        pass

    def predict(self, X):
        pass
```

```

self.feature_importance = {}
self.scaler = None
self.best_model = None
self.best_model_name = None

def prepare_features(self, X, y=None, scaling_method='standard'):
    """
    Подготовка и масштабирование признаков

    Args:
        X (DataFrame): Признаки
        y (Series): Целевая переменная
        scaling_method (str): Метод масштабирования ('standard', 'robust', 'none')
    """
    X_processed = X.copy()

    # Обработка пропущенных значений
    X_processed = X_processed.fillna(X_processed.mean())

    # Масштабирование признаков
    if scaling_method == 'standard':
        self.scaler = StandardScaler()
        X_processed = self.scaler.fit_transform(X_processed)
    elif scaling_method == 'robust':
        self.scaler = RobustScaler()
        X_processed = self.scaler.fit_transform(X_processed)

    return X_processed

def feature_selection(self, X, y, k=10):
    """
    Отбор наиболее важных признаков

    Args:
        X (DataFrame): Признаки
        y (Series): Целевая переменная
        k (int): Количество лучших признаков для отбора
    """
    selector = SelectKBest(score_func=f_regression, k=min(k, X.shape[1]))
    X_selected = selector.fit_transform(X, y)

    # Получаем важность признаков
    feature_scores = pd.DataFrame({
        'feature': X.columns,
        'score': selector.scores_,
        'p_value': selector.pvalues_
    })
    feature_scores = feature_scores.sort_values('score', ascending=False)

    selected_features = feature_scores.head(k)['feature'].tolist()

    return X_selected, selected_features, feature_scores

def initialize_models(self):
    """
    Инициализация различных моделей машинного обучения
    """
    self.models = {
        'Linear Regression': LinearRegression(),
        'Ridge Regression': Ridge(alpha=1.0, random_state=self.random_state),
        'Lasso Regression': Lasso(alpha=0.1, random_state=self.random_state),
        'Random Forest': RandomForestRegressor(
            n_estimators=200,
            max_depth=10,
            min_samples_split=5,
            random_state=self.random_state
        ),
        'Gradient Boosting': GradientBoostingRegressor(
            n_estimators=200,
            learning_rate=0.1,
            max_depth=5,
            random_state=self.random_state
        ),
        'Support Vector Regression': SVR(kernel='rbf', C=1.0),
        'Neural Network': MLPRegressor(
            hidden_layer_sizes=(100, 50),
            max_iter=1000,
            random_state=self.random_state
        )
    }

def train_models(self, X_train, y_train, X_test, y_test, cv_folds=5):

```

```

"""
Обучение различных моделей машинного обучения с кросс-валидацией

Args:
    X_train, X_test: Признаки для обучения и тестирования
    y_train, y_test: Целевые переменные
    cv_folds (int): Количество фолдов для кросс-валидации
"""

self.initialize_models()
self.results = {}

# Time Series Split для временных рядов
tscv = TimeSeriesSplit(n_splits=cv_folds)

for name, model in self.models.items():
    print(f"Обучение модели: {name}")

    try:
        # Обучение модели
        model.fit(X_train, y_train)

        # Предсказания
        y_pred = model.predict(X_test)

        # Метрики качества
        mae = mean_absolute_error(y_test, y_pred)
        rmse = np.sqrt(mean_squared_error(y_test, y_pred))
        r2 = r2_score(y_test, y_pred)

        # Кросс-валидация
        cv_scores = cross_val_score(model, X_train, y_train,
                                     cv=tscv, scoring='r2', n_jobs=-1)
        cv_mean = cv_scores.mean()
        cv_std = cv_scores.std()

        # Точность в ±10%
        accuracy_10_percent = self.calculate_accuracy_within_percentage(y_test, y_pred, 10)

        # Сохранение результатов
        self.results[name] = {
            'model': model,
            'predictions': y_pred,
            'mae': mae,
            'rmse': rmse,
            'r2': r2,
            'cv_mean': cv_mean,
            'cv_std': cv_std,
            'accuracy_10_percent': accuracy_10_percent
        }

        # Важность признаков (для соответствующих моделей)
        if hasattr(model, 'feature_importances_'):
            self.feature_importance[name] = {
                'importances': model.feature_importances_,
                'feature_names': X_train.columns if hasattr(X_train, 'columns') else list(range(X_train.shape[1]))
            }

        print(f" R2: {r2:.4f}, MAE: {mae:.2f}, ±10% точность: {accuracy_10_percent:.2f}%")

    except Exception as e:
        print(f" Ошибка при обучении {name}: {e}")
        continue

    # Определение лучшей модели
    self._select_best_model()

return self.results

def calculate_accuracy_within_percentage(self, y_true, y_pred, percentage):
    """
    Расчет точности предсказаний в пределах заданного процента
    """
    percentage_decimal = percentage / 100.0
    within_range = np.abs(y_true - y_pred) <= (np.abs(y_true) * percentage_decimal)
    accuracy = np.mean(within_range) * 100
    return accuracy

def _select_best_model(self):
    """Выбор лучшей модели на основе R2 score"""
    if not self.results:
        return None

```

```

best_r2 = -np.inf
best_model_name = None

for name, result in self.results.items():
    if result['r2'] > best_r2:
        best_r2 = result['r2']
        best_model_name = name

self.best_model_name = best_model_name
self.best_model = self.results[best_model_name]['model']

return best_model_name

def hyperparameter_tuning(self, X_train, y_train, model_name='Random Forest'):
    """
    Настройка гиперпараметров для выбранной модели
    """
    if model_name not in self.models:
        print(f"Модель {model_name} не найдена")
        return None

    param_grids = {
        'Random Forest': {
            'n_estimators': [100, 200, 300],
            'max_depth': [5, 10, 15, None],
            'min_samples_split': [2, 5, 10]
        },
        'Gradient Boosting': {
            'n_estimators': [100, 200, 300],
            'learning_rate': [0.01, 0.1, 0.2],
            'max_depth': [3, 5, 7]
        },
        'Ridge Regression': {
            'alpha': [0.1, 1.0, 10.0, 100.0]
        },
        'Lasso Regression': {
            'alpha': [0.001, 0.01, 0.1, 1.0]
        }
    }

    if model_name not in param_grids:
        print(f"Сетка параметров для {model_name} не определена")
        return None

    print(f"Настройка гиперпараметров для {model_name}...")

    grid_search = GridSearchCV(
        self.models[model_name],
        param_grids[model_name],
        cv=TimeSeriesSplit(n_splits=3),
        scoring='r2',
        n_jobs=-1
    )

    grid_search.fit(X_train, y_train)

    print(f"Лучшие параметры: {grid_search.best_params_}")
    print(f"Лучший R2: {grid_search.best_score_.4f}")

    return grid_search.best_estimator_

def analyze_feature_importance(self, feature_names):
    """
    Анализ важности признаков для моделей
    """
    importance_analysis = {}

    for model_name, importance_data in self.feature_importance.items():
        importances = importance_data['importances']
        features = importance_data['feature_names']

        # Создаем DataFrame с важностью признаков
        feature_importance_df = pd.DataFrame({
            'feature': features,
            'importance': importances
        }).sort_values('importance', ascending=False)

        importance_analysis[model_name] = feature_importance_df

    # Вывод топ-5 признаков
    print(f"\n{model_name} - Топ-5 важных признаков:")
    for i, row in feature_importance_df.head().iterrows():

```

```

print(f" {row['feature']}: {row['importance']:.3f} ({row['importance']*100:.1f}%)")

# Сводная важность признаков (усредненная по всем моделям)
if importance_analysis:
    all_importances = []
    for model_name, df in importance_analysis.items():
        df = df.copy()
        df['model'] = model_name
        all_importances.append(df)

    combined_importance = pd.concat(all_importances)
    avg_importance = combined_importance.groupby('feature')['importance'].mean().sort_values(ascending=False)

    print(f"\nСредняя важность признаков по всем моделям:")
    for feature, importance in avg_importance.head(10).items():
        print(f" {feature}: {importance:.3f} ({importance*100:.1f}%)")

    return importance_analysis, avg_importance

return importance_analysis, None

def predict_future_trends(self, X_future, model_name=None):
    """
    Прогнозирование будущих трендов
    """
    if model_name is None:
        model_name = self.best_model_name

    if model_name not in self.results:
        print(f"Модель {model_name} не найдена в результатах")
        return None

    model = self.results[model_name]['model']
    predictions = model.predict(X_future)

    return predictions

def visualize_results(self, y_test, feature_names=None):
    """
    Визуализация результатов прогнозирования
    """
    if not self.results:
        print("Нет результатов для визуализации")
        return None

    fig, axes = plt.subplots(2, 2, figsize=(15, 12))
    fig.suptitle('Анализ прогнозирования финансовых трендов', fontsize=16, fontweight='bold')

    # 1. Сравнение метрик моделей
    model_names = list(self.results.keys())
    r2_scores = [self.results[name]['r2'] for name in model_names]
    accuracy_scores = [self.results[name]['accuracy_10_percent'] for name in model_names]

    x_pos = np.arange(len(model_names))
    width = 0.35

    axes[0, 0].bar(x_pos - width/2, r2_scores, width, label='R2 Score', alpha=0.7)
    axes[0, 0].bar(x_pos + width/2, accuracy_scores, width, label='±10% Accuracy', alpha=0.7)
    axes[0, 0].set_xlabel('Модели')
    axes[0, 0].set_ylabel('Метрики')
    axes[0, 0].set_title('Сравнение производительности моделей')
    axes[0, 0].set_xticks(x_pos)
    axes[0, 0].set_xticklabels(model_names, rotation=45)
    axes[0, 0].legend()
    axes[0, 0].grid(True, alpha=0.3)

    # 2. Прогнозы лучшей модели vs Фактические значения
    best_model_name = self.best_model_name
    best_predictions = self.results[best_model_name]['predictions']

    axes[0, 1].scatter(y_test, best_predictions, alpha=0.6)
    axes[0, 1].plot([y_test.min(), y_test.max()], [y_test.min(), y_test.max()], 'r--', lw=2)
    axes[0, 1].set_xlabel('Фактические значения')
    axes[0, 1].set_ylabel('Прогнозируемые значения')
    axes[0, 1].set_title(f'Прогнозы vs Факт ({best_model_name})\nR2 = {self.results[best_model_name]["r2"]:.4f}')
    axes[0, 1].grid(True, alpha=0.3)

    # 3. Важность признаков (если доступно)
    if self.feature_importance and feature_names is not None:
        if best_model_name in self.feature_importance:
            importances = self.feature_importance[best_model_name]['importances']
            features = self.feature_importance[best_model_name]['feature_names']

```

```

importance_df = pd.DataFrame({
    'feature': features,
    'importance': importances
}).sort_values('importance', ascending=True).tail(10)

axes[1, 0].barh(importance_df['feature'], importance_df['importance'])
axes[1, 0].set_xlabel('Важность признака')
axes[1, 0].set_title(f'Топ-10 важных признаков ({best_model_name})')
axes[1, 0].grid(True, alpha=0.3)

# 4. Распределение ошибок
errors = best_predictions - y_test
axes[1, 1].hist(errors, bins=30, alpha=0.7, color='orange', edgecolor='black')
axes[1, 1].axvline(errors.mean(), color='red', linestyle='--',
                   label=f'Средняя ошибка: {errors.mean():.2f}')
axes[1, 1].set_xlabel('Ошибка прогноза')
axes[1, 1].set_ylabel('Частота')
axes[1, 1].set_title('Распределение ошибок прогнозирования')
axes[1, 1].legend()
axes[1, 1].grid(True, alpha=0.3)

plt.tight_layout()
return fig
}

def generate_report(self):
    """
    Генерация детального отчета по прогнозированию
    """
    if not self.results:
        return None

    report = {
        'best_model': {
            'name': self.best_model_name,
            'metrics': self.results[self.best_model_name]
        },
        'all_models': {},
        'summary': {
            'total_models_tested': len(self.results),
            'best_r2_score': self.results[self.best_model_name]['r2'],
            'best_accuracy_10_percent': self.results[self.best_model_name]['accuracy_10_percent'],
            'average_predicted_return': np.mean(self.results[self.best_model_name]['predictions'])
        }
    }

    for name, result in self.results.items():
        report['all_models'][name] = {
            'r2': result['r2'],
            'mae': result['mae'],
            'rmse': result['rmse'],
            'accuracy_10_percent': result['accuracy_10_percent'],
            'cv_mean': result['cv_mean'],
            'cv_std': result['cv_std']
        }

    return report
}

# Пример использования
def generate_sample_financial_data(n_samples=1000):
    """
    Генерация тестовых финансовых данных
    """
    np.random.seed(42)

    # Макроэкономические показатели
    data = {
        'stock_index': np.random.normal(100, 20, n_samples),
        'deposit_volume': np.random.lognormal(10, 1, n_samples),
        'mobile_payments': np.random.exponential(1000, n_samples),
        'interest_rate': np.random.uniform(1, 10, n_samples),
        'inflation_rate': np.random.uniform(1, 8, n_samples),
        'gdp_growth': np.random.normal(3, 1, n_samples),
        'unemployment_rate': np.random.uniform(3, 10, n_samples),
        'consumer_confidence': np.random.normal(50, 10, n_samples),
        'exchange_rate': np.random.normal(75, 5, n_samples),
        'market_volatility': np.random.exponential(2, n_samples)
    }

    # Целевая переменная - доходность финтех-компаний
    # (синтетическая зависимость от признаков)

```

```
X = pd.DataFrame(data)
coefficients = np.array([0.3, 0.25, 0.2, -0.15, -0.1, 0.12, -0.08, 0.05, 0.03, -0.02])
noise = np.random.normal(0, 50, n_samples)

y = X.dot(coefficients) + noise + 500

return X, y

if __name__ == "__main__":
    # Генерация тестовых данных
    print("Генерация финансовых данных...")
    X, y = generate_sample_financial_data(1000)

    # Разделение на train/test
    split_idx = int(0.8 * len(X))
    X_train, X_test = X.iloc[:split_idx], X.iloc[split_idx:]
    y_train, y_test = y[:split_idx], y[split_idx:]

    # Создание и обучение прогнозировщика
    predictor = FinancialTrendPredictor()

    # Подготовка данных
    X_train_processed = predictor.prepare_features(X_train, scaling_method='standard')
    X_test_processed = predictor.prepare_features(X_test, scaling_method='standard')

    # Обучение моделей
    print("\nОбучение моделей машинного обучения...")
    results = predictor.train_models(X_train_processed, y_train, X_test_processed, y_test)

    # Анализ важности признаков
    print("\nАнализ важности признаков...")
    importance_analysis, avg_importance = predictor.analyze_feature_importance(X.columns)

    # Генерация отчета
    report = predictor.generate_report()

    # Вывод результатов
    print("\n" + "*60)
    print("ОТЧЕТ ПО ПРОГНОЗИРОВАНИЮ ФИНАНСОВЫХ ТРЕНДОВ")
    print("*60)
    print(f"Лучшая модель: {report['best_model']['name']}")
    print(f"R2 лучшей модели: {report['best_model']['metrics']['r2']:.4f}")
    print(f"Точность предсказаний в ±10%: {report['best_model']['metrics']['accuracy_10_percent']:.2f}%")
    print(f"Средняя прогнозная доходность: {report['summary']['average_predicted_return']:.0f} py6.")

    print(f"\nСравнение моделей:")
    for model_name, metrics in report['all_models'].items():
        print(f" {model_name}: R2 = {metrics['r2']:.4f}, ±10% = {metrics['accuracy_10_percent']:.2f}%")

    # Визуализация
    fig = predictor.visualize_results(y_test, feature_names=X.columns)
    plt.show()
```

Генерация финансовых данных...

```
Обучение моделей машинного обучения...
Обучение модели: Linear Regression
R2: 0.9930, MAE: 841.26, ±10% точность: 43.00%
Обучение модели: Ridge Regression
R2: 0.9928, MAE: 847.53, ±10% точность: 43.00%
Обучение модели: Lasso Regression
R2: 0.9930, MAE: 841.32, ±10% точность: 43.00%
Обучение модели: Random Forest
R2: 0.9805, MAE: 924.40, ±10% точность: 42.50%
Обучение модели: Gradient Boosting
R2: 0.9919, MAE: 870.15, ±10% точность: 41.50%
Обучение модели: Support Vector Regression
R2: -0.0648, MAE: 6093.37, ±10% точность: 8.00%
Обучение модели: Neural Network
R2: 0.9933, MAE: 829.93, ±10% точность: 45.00%
```

Анализ важности признаков...

Random Forest - Топ-5 важных признаков:

```
1.0: 0.995 (99.5%)
6.0: 0.002 (0.2%)
2.0: 0.000 (0.0%)
7.0: 0.000 (0.0%)
0.0: 0.000 (0.0%)
```

Gradient Boosting - Топ-5 важных признаков:

```
1.0: 0.999 (99.9%)
2.0: 0.000 (0.0%)
8.0: 0.000 (0.0%)
4.0: 0.000 (0.0%)
3.0: 0.000 (0.0%)
```

Средняя важность признаков по всем моделям:

```
1: 0.997 (99.7%)
6: 0.001 (0.1%)
2: 0.000 (0.0%)
8: 0.000 (0.0%)
7: 0.000 (0.0%)
0: 0.000 (0.0%)
3: 0.000 (0.0%)
4: 0.000 (0.0%)
5: 0.000 (0.0%)
9: 0.000 (0.0%)
```

=====

ОТЧЕТ ПО ПРОГНОЗИРОВАНИЮ ФИНАНСОВЫХ ТРЕНДОВ

=====

Лучшая модель: Neural Network

R² лучшей модели: 0.9933

Точность предсказаний в ±10%: 45.00%

Средняя прогнозная доходность: 10476 руб.

Сравнение моделей:

Linear Regression: R² = 0.9930, ±10% = 43.00%

Ridge Regression: R² = 0.9928, ±10% = 43.00%

Lasso Regression: R² = 0.9930, ±10% = 43.00%

задание Random Forest: R² = 0.9805, ±10% = 42.50%

Gradient Boosting: R² = 0.9919, ±10% = 41.50%

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.ensemble import RandomForestClassifier, GradientBoostingClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.svm import SVC
from sklearn.metrics import accuracy_score, classification_report, confusion_matrix, roc_auc_score
from sklearn.preprocessing import StandardScaler, LabelEncoder
from sklearn.model_selection import cross_val_score, train_test_split
import warnings
warnings.filterwarnings('ignore')

class IoTFinanceAnalyzer:
    """
    Расширенный класс для анализа данных IoT устройств в финансовых услугах
    """

    def __init__(self, iot_data):
        """
        Инициализация с данными IoT устройств

        Args:
            iot_data (DataFrame): Данные IoT устройств
        """
        self.iot_data = iot_data
```

```

self.risk_scores = None
self.models = {}
self.scaler = StandardScaler()
self.feature_importance = {}
self.user_segments = None

def preprocess_iot_data(self):
    """
    Предобработка и обогащение IoT данных
    """
    df = self.iot_data.copy()

    # Заполнение пропущенных значений
    numeric_columns = df.select_dtypes(include=[np.number]).columns
    df[numeric_columns] = df[numeric_columns].fillna(df[numeric_columns].mean())

    # Кодирование категориальных переменных
    categorical_columns = df.select_dtypes(include=['object']).columns
    for col in categorical_columns:
        if col != 'user_id':
            le = LabelEncoder()
            df[col] = le.fit_transform(df[col].astype(str))

    # Создание агрегированных признаков по пользователям
    user_aggregated = self._create_user_aggregates(df)

    return df, user_aggregated

def _create_user_aggregates(self, df):
    """
    Создание агрегированных признаков на уровне пользователей
    """

    # Сначала создаем базовые агрегации по пользователям
    basic_agg = df.groupby('user_id').agg({
        'device_type': 'count'
    }).rename(columns={'device_type': 'total_devices'})

    # Агрегации по типам устройств
    device_aggregates = {}

    # Smart Home агрегации
    smart_home_data = df[df['device_type'] == 'smart_home']
    if not smart_home_data.empty:
        smart_home_agg = smart_home_data.groupby('user_id').agg({
            'security_events': ['mean', 'sum'],
            'energy_consumption': ['mean', 'std'],
            'active_devices': 'mean'
        })
        smart_home_agg.columns = [
            'smart_home_avg_security_events', 'smart_home_total_security_events',
            'smart_home_avg_energy_consumption', 'smart_home_energy_consumption_std',
            'smart_home_active_devices_count'
        ]
        device_aggregates['smart_home'] = smart_home_agg

    # Wearable агрегации
    wearable_data = df[df['device_type'] == 'wearable']
    if not wearable_data.empty:
        wearable_agg = wearable_data.groupby('user_id').agg({
            'daily_steps': ['mean', 'std'],
            'sleep_quality': 'mean',
            'stress_level': 'mean'
        })
        wearable_agg.columns = [
            'wearable_avg_daily_steps', 'wearable_activity_consistency',
            'wearable_sleep_quality_score', 'wearable_stress_level_avg'
        ]
        device_aggregates['wearable'] = wearable_agg

    # Car Telematics агрегации
    car_data = df[df['device_type'] == 'car_telematics']
    if not car_data.empty:
        car_agg = car_data.groupby('user_id').agg({
            'harsh_braking': 'mean',
            'speeding_events': 'mean',
            'km_driven': 'sum',
            'night_driving': 'mean',
            'maintenance_due': 'mean'
        })
        car_agg.columns = [
            'car_telematics_avg_harsh_braking', 'car_telematics_avg_speeding_events',
            'car_telematics_total_km_driven', 'car_telematics_night_driving_ratio',
        ]

```

```

        'car_telematics_maintenance_regularity'
    ]
device_aggregates['car_telematics'] = car_agg

# POS Terminal агрегации
pos_data = df[df['device_type'] == 'pos_terminal']
if not pos_data.empty:
    pos_agg = pos_data.groupby('user_id').agg({
        'transaction_count': 'mean',
        'transaction_value': ['mean', 'std'],
        'business_hours': 'mean'
    })
    pos_agg.columns = [
        'pos_terminal_daily_transaction_count', 'pos_terminal_avg_transaction_value',
        'pos_terminal_transaction_volume_std', 'pos_terminal_business_hours_activity'
    ]
    device_aggregates['pos_terminal'] = pos_agg

# Объединение всех агрегаций
user_aggregated = basic_agg
for device_type, agg_data in device_aggregates.items():
    user_aggregated = user_aggregated.join(agg_data, how='left')

# Заполнение пропущенных значений
user_aggregated = user_aggregated.fillna(0)

return user_aggregated

def calculate_comprehensive_risk_scores(self, user_aggregated):
    """
    Расчет комплексных скорингов риска на основе IoT данных
    """
    risk_categories = {}

    # 1. Риск безопасности жилья (Smart Home)
    security_risk = np.zeros(len(user_aggregated))
    if 'smart_home_avg_security_events' in user_aggregated.columns:
        security_risk = np.where(
            user_aggregated['smart_home_avg_security_events'] > 3, 20,
            np.where(user_aggregated['smart_home_avg_security_events'] > 1, 10, 0)
        )
    risk_categories['security_risk'] = security_risk

    # 2. Риск вождения (Car Telematics)
    driving_risk_components = np.zeros(len(user_aggregated))
    if 'car_telematics_avg_harsh_braking' in user_aggregated.columns:
        braking_risk = np.where(
            user_aggregated['car_telematics_avg_harsh_braking'] > 2, 15,
            np.where(user_aggregated['car_telematics_avg_harsh_braking'] > 1, 8, 0)
        )
        driving_risk_components += braking_risk

    if 'car_telematics_avg_speeding_events' in user_aggregated.columns:
        speeding_risk = np.where(
            user_aggregated['car_telematics_avg_speeding_events'] > 5, 12,
            np.where(user_aggregated['car_telematics_avg_speeding_events'] > 2, 6, 0)
        )
        driving_risk_components += speeding_risk

    if 'car_telematics_night_driving_ratio' in user_aggregated.columns:
        night_driving_risk = np.where(
            user_aggregated['car_telematics_night_driving_ratio'] > 0.3, 10, 0
        )
        driving_risk_components += night_driving_risk

    risk_categories['driving_risk'] = np.minimum(driving_risk_components, 30)

    # 3. Риск здоровья (Wearable)
    health_risk_components = np.zeros(len(user_aggregated))
    if 'wearable_avg_daily_steps' in user_aggregated.columns:
        activity_risk = np.where(
            user_aggregated['wearable_avg_daily_steps'] < 5000, 8,
            np.where(user_aggregated['wearable_avg_daily_steps'] < 8000, 4, 0)
        )
        health_risk_components += activity_risk

    if 'wearable_sleep_quality_score' in user_aggregated.columns:
        sleep_risk = np.where(
            user_aggregated['wearable_sleep_quality_score'] < 60, 6,
            np.where(user_aggregated['wearable_sleep_quality_score'] < 75, 3, 0)
        )
        health_risk_components += sleep_risk

    risk_categories['health_risk'] = np.minimum(health_risk_components, 30)

```

```

if 'wearable_stress_level_avg' in user_aggregated.columns:
    stress_risk = np.where(
        user_aggregated['wearable_stress_level_avg'] > 7, 10, 0
    )
    health_risk_components += stress_risk

risk_categories['health_risk'] = np.minimum(health_risk_components, 20)

# 4. Бизнес-риск (POS Terminal)
business_risk_components = np.zeros(len(user_aggregated))
if 'pos_terminal_transaction_volume_std' in user_aggregated.columns:
    volatility_risk = np.where(
        user_aggregated['pos_terminal_transaction_volume_std'] > 1000, 15, 0
    )
    business_risk_components += volatility_risk

if 'pos_terminal_business_hours_activity' in user_aggregated.columns:
    activity_risk = np.where(
        user_aggregated['pos_terminal_business_hours_activity'] < 0.3, 10, 0
    )
    business_risk_components += activity_risk

risk_categories['business_risk'] = business_risk_components

# Общий риск скоринг
total_risk = np.zeros(len(user_aggregated))
for risk_array in risk_categories.values():
    total_risk += risk_array

# Нормализация до 100 баллов
max_possible_risk = 20 + 30 + 25 # Максимальные значения по категориям
normalized_risk = (total_risk / max_possible_risk) * 100

risk_results = pd.DataFrame({
    'user_id': user_aggregated.index,
    'total_risk_score': normalized_risk,
    'risk_category': np.where(normalized_risk >= 70, 'High',
                              np.where(normalized_risk >= 40, 'Medium', 'Low'))
})

```

Добавление категориальных рисков

```

for category, scores in risk_categories.items():
    risk_results[category] = scores

self.risk_scores = risk_results
return risk_results

```

```

def segment_users(self, user_aggregated):
    """
    Сегментация пользователей на основе IoT поведения
    """
    segments = []

    for user_id in user_aggregated.index:
        user_data = user_aggregated.loc[user_id]

        # Определение сегмента на основе поведения
        segment = 'Standard_User'

        if 'car_telematics_total_km_driven' in user_aggregated.columns:
            if user_data.get('car_telematics_total_km_driven', 0) > 1000:
                segment = 'High_Mobility'
            elif user_data.get('pos_terminal_daily_transaction_count', 0) > 20:
                segment = 'Business_Owner'
            elif user_data.get('wearable_avg_daily_steps', 0) > 10000:
                segment = 'Active_Lifestyle'

        segments.append({'user_id': user_id, 'segment': segment})

    self.user_segments = pd.DataFrame(segments)
    return self.user_segments

```

```

def analyze_iot_patterns(self, user_aggregated):
    """
    Анализ паттернов поведения на основе IoT данных
    """
    analysis = {}

    # Анализ по типам устройств
    device_analysis = {}

```

```

# Smart Home анализ
if 'smart_home_avg_security_events' in user_aggregated.columns:
    high_security_users = (user_aggregated['smart_home_avg_security_events'] > 2).sum()
    device_analysis['smart_home_high_risk'] = int(high_security_users)

# Car Telematics анализ
if 'car_telematics_avg_harsh_braking' in user_aggregated.columns:
    risky_drivers = (user_aggregated['car_telematics_avg_harsh_braking'] > 1).sum()
    device_analysis['risky_drivers'] = int(risky_drivers)

# POS Terminal анализ
if 'pos_terminal_daily_transaction_count' in user_aggregated.columns:
    avg_business_activity = user_aggregated['pos_terminal_daily_transaction_count'].mean()
    device_analysis['avg_business_activity'] = float(avg_business_activity)

analysis['device_patterns'] = device_analysis

return analysis

def generate_recommendations(self):
    """
    Генерация персонализированных рекомендаций
    """
    if self.risk_scores is None or self.user_segments is None:
        raise ValueError("Сначала выполните расчет рисков и сегментацию пользователей")

    recommendations = []

    for _, user in self.risk_scores.iterrows():
        user_id = user['user_id']
        risk_level = user['risk_category']
        user_segment = self.user_segments[self.user_segments['user_id'] == user_id]['segment'].iloc[0]

        recs = []

        if risk_level == 'High':
            recs.extend([
                "Рекомендуется консультация финансового советника",
                "Рассмотреть дополнительные страховые продукты",
                "Мониторинг финансовой активности"
            ])
        elif risk_level == 'Medium':
            recs.extend([
                "Регулярный обзор финансовых привычек",
                "Образовательные материалы по финансовой грамотности"
            ])

        # Рекомендации по сегментам
        if user_segment == 'High_Mobility':
            recs.append("Специальные авто-страховые предложения")
        elif user_segment == 'Business_Owner':
            recs.append("Бизнес-кредитные продукты с мониторингом активности")
        elif user_segment == 'Active_Lifestyle':
            recs.append("Страхование здоровья со скидкой за активность")

        recommendations.append({
            'user_id': user_id,
            'risk_level': risk_level,
            'segment': user_segment,
            'recommendations': recs
        })

    return pd.DataFrame(recommendations)

def visualize_analysis(self, user_aggregated):
    """
    Визуализация анализа IoT данных
    """
    if self.risk_scores is None:
        raise ValueError("Сначала выполните расчет рисков")

    fig, axes = plt.subplots(2, 3, figsize=(18, 12))
    fig.suptitle('Анализ IoT данных для финансовых услуг', fontsize=16, fontweight='bold')

    # 1. Распределение рисков
    risk_dist = self.risk_scores['risk_category'].value_counts()
    colors = ['green', 'orange', 'red']
    axes[0, 0].pie(risk_dist.values, labels=risk_dist.index, autopct='%1.1f%%',
                  colors=colors, startangle=90)
    axes[0, 0].set_title('Распределение пользователей по уровню риска')

    # 2. Средние показатели по устройствам

```

```

device_metrics = {}
for col in user_aggregated.columns:
    if 'avg' in col or 'mean' in col:
        device_metrics[col] = user_aggregated[col].mean()

if device_metrics:
    metrics_df = pd.DataFrame(list(device_metrics.items()), columns=['Metric', 'Value'])
    metrics_df = metrics_df.head(8) # Ограничиваем для читаемости

    axes[0, 1].barh(metrics_df['Metric'], metrics_df['Value'])
    axes[0, 1].set_title('Средние показатели IoT устройств')
    axes[0, 1].set_xlabel('Значение')

# 3. Корреляция рисков
risk_columns = [col for col in self.risk_scores.columns if 'risk' in col and col != 'total_risk_score']
if len(risk_columns) > 1:
    risk_correlation = self.risk_scores[risk_columns].corr()
    sns.heatmap(risk_correlation, annot=True, cmap='coolwarm', center=0,
                ax=axes[0, 2], cbar_kws={'label': 'Корреляция'})
    axes[0, 2].set_title('Корреляция категорий риска')
else:
    axes[0, 2].text(0.5, 0.5, 'Недостаточно данных\\пдля корреляционного анализа',
                    ha='center', va='center', transform=axes[0, 2].transAxes)
    axes[0, 2].set_title('Корреляция категорий риска')

# 4. Распределение бизнес-активности
if 'pos_terminal_daily_transaction_count' in user_aggregated.columns:
    axes[1, 0].hist(user_aggregated['pos_terminal_daily_transaction_count'],
                   bins=20, alpha=0.7, color='blue', edgecolor='black')
    axes[1, 0].axvline(user_aggregated['pos_terminal_daily_transaction_count'].mean(),
                       color='red', linestyle='--',
                       label=f'Среднее: {user_aggregated["pos_terminal_daily_transaction_count"].mean():.1f}')
    axes[1, 0].set_xlabel('Транзакций в день')
    axes[1, 0].set_ylabel('Количество пользователей')
    axes[1, 0].set_title('Распределение бизнес-активности')
    axes[1, 0].legend()
else:
    axes[1, 0].text(0.5, 0.5, 'Нет данных по\\бизнес-активности',
                    ha='center', va='center', transform=axes[1, 0].transAxes)
    axes[1, 0].set_title('Распределение бизнес-активности')

# 5. Распределение рисков вождения
if 'car_telematics_avg_harsh_braking' in user_aggregated.columns:
    axes[1, 1].hist(user_aggregated['car_telematics_avg_harsh_braking'],
                   bins=20, alpha=0.7, color='red', edgecolor='black')
    axes[1, 1].axvline(user_aggregated['car_telematics_avg_harsh_braking'].mean(),
                       color='blue', linestyle='--',
                       label=f'Среднее: {user_aggregated["car_telematics_avg_harsh_braking"].mean():.1f}')
    axes[1, 1].set_xlabel('Резкие торможения в день')
    axes[1, 1].set_ylabel('Количество пользователей')
    axes[1, 1].set_title('Распределение рисков вождения')
    axes[1, 1].legend()
else:
    axes[1, 1].text(0.5, 0.5, 'Нет данных по\\телеинформатике автомобиля',
                    ha='center', va='center', transform=axes[1, 1].transAxes)
    axes[1, 1].set_title('Распределение рисков вождения')

# 6. Сегментация пользователей
if self.user_segments is not None:
    segment_dist = self.user_segments['segment'].value_counts()
    axes[1, 2].pie(segment_dist.values, labels=segment_dist.index, autopct='%1.1f%%',
                  startangle=90)
    axes[1, 2].set_title('Сегментация пользователей')
else:
    axes[1, 2].text(0.5, 0.5, 'Сегментация\\пне выполнена',
                    ha='center', va='center', transform=axes[1, 2].transAxes)
    axes[1, 2].set_title('Сегментация пользователей')

plt.tight_layout()
return fig

def generate_comprehensive_report(self):
    """
    Генерация комплексного отчета по анализу IoT данных
    """
    # Предобработка данных
    processed_data, user_aggregated = self.preprocess_iot_data()

    # Расчет рисков
    risk_scores = self.calculate_comprehensive_risk_scores(user_aggregated)

    # Сегментация пользователей

```

```

user_segments = self.segment_users(user_aggregated)

# Анализ паттернов
patterns_analysis = self.analyze_iot_patterns(user_aggregated)

# Рекомендации
recommendations = self.generate_recommendations()

# Формирование отчета
report = {
    'summary': {
        'total_iot_records': len(self.iot_data),
        'total_users': len(user_aggregated),
        'high_risk_users': (risk_scores['risk_category'] == 'High').sum(),
        'avg_business_activity': patterns_analysis['device_patterns'].get('avg_business_activity', 0),
    },
    'risk_analysis': {
        'risk_distribution': risk_scores['risk_category'].value_counts().to_dict(),
        'avg_risk_score': risk_scores['total_risk_score'].mean(),
        'high_risk_ratio': (risk_scores['risk_category'] == 'High').mean() * 100
    },
    'user_segmentation': user_segments['segment'].value_counts().to_dict(),
    'device_analysis': patterns_analysis['device_patterns'],
    'recommendations_sample': recommendations.head(10).to_dict('records')
}

return report
}

# Исправленная генерация тестовых данных
def generate_sample_iot_data(n_users=200, n_records=60000):
    """
    Генерация тестовых IoT данных
    """
    np.random.seed(42)

    user_ids = [f'USER_{i:05d}' for i in range(1, n_users + 1)]
    device_types = ['smart_home', 'wearable', 'car_telematics', 'pos_terminal']

    records = []

    for i in range(n_records):
        user_id = np.random.choice(user_ids)
        device_type = np.random.choice(device_types, p=[0.3, 0.25, 0.25, 0.2])

        record = {
            'user_id': user_id,
            'device_type': device_type,
            'timestamp': pd.Timestamp.now() - pd.Timedelta(minutes=np.random.randint(1, 10080))
        }

        # Генерация device-specific данных
        if device_type == 'smart_home':
            record.update({
                'security_events': np.random.poisson(1.5),
                'energy_consumption': np.random.normal(50, 15),
                'active_devices': np.random.randint(1, 10)
            })
        elif device_type == 'wearable':
            record.update({
                'daily_steps': np.random.normal(8000, 3000),
                'sleep_quality': np.random.randint(40, 95),
                'stress_level': np.random.randint(1, 10),
                'heart_rate': np.random.normal(70, 10)
            })
        elif device_type == 'car_telematics':
            record.update({
                'harsh_braking': np.random.poisson(0.8),
                'speeding_events': np.random.poisson(2),
                'km_driven': np.random.exponential(50),
                'night_driving': np.random.random() > 0.7,
                'maintenance_due': np.random.random() > 0.9
            })
        elif device_type == 'pos_terminal':
            record.update({
                'transaction_count': np.random.poisson(25),
                'transaction_value': np.random.lognormal(6, 1),
                'business_hours': np.random.random() > 0.3
            })

        records.append(record)
}

```

```

return pd.DataFrame(records)

if __name__ == "__main__":
    # Генерация тестовых данных
    print("Генерация IoT данных...")
    iot_data = generate_sample_iot_data(n_records=1000) # Уменьшим для скорости

    print(f"Сгенерировано {len(iot_data)} записей")
    print(f"Уникальных пользователей: {iot_data['user_id'].nunique()}")
    print(f"Типы устройств: {iot_data['device_type'].value_counts().to_dict()}")

    # Создание анализатора
    analyzer = IoTFinanceAnalyzer(iot_data)

    # Генерация отчета
    print("\nАнализ данных...")
    report = analyzer.generate_comprehensive_report()

    # Вывод результатов
    print("\n" + "*60)
    print("ОТЧЕТ ПО АНАЛИЗУ IoT ДАННЫХ В ФИНАНСАХ")
    print("*60)
    print(f"Общее количество IoT записей: {report['summary']['total_iot_records']}:")
    print(f"Количество пользователей: {report['summary']['total_users']} ")
    print(f"Высокорисковые пользователи: {report['summary']['high_risk_users']} из {report['summary']['total_users']} ")

    if 'avg_business_activity' in report['summary']:
        print(f"Средняя бизнес-активность: {report['summary']['avg_business_activity']:.1f} транзакций/день")

    print(f"\nРаспределение рисков:")
    for risk_level, count in report['risk_analysis']['risk_distribution'].items():
        percentage = (count / report['summary']['total_users']) * 100
        print(f" {risk_level}: {count} пользователей ({percentage:.1f}%)")

    print(f"\nСегменты пользователей:")
    for segment, count in report['user_segmentation'].items():
        percentage = (count / report['summary']['total_users']) * 100
        print(f" {segment}: {count} пользователей ({percentage:.1f}%)")

    # Визуализация
    print("\nСоздание визуализаций...")
    processed_data, user_aggregated = analyzer.preprocess_iot_data()
    fig = analyzer.visualize_analysis(user_aggregated)
    plt.show()

    # Пример рекомендаций
    print("\nПример рекомендаций для пользователей:")

```

```

Генерация IoT данных...
Сгенерировано 1000 записей
Уникальных пользователей: 199
Типы устройств: {'smart_home': 292, 'car_telematics': 258, 'wearable': 255, 'pos_terminal': 199}
Анализ данных...

```