Сравнение решений задачи классификации узлов сети с использованием различных методов векторных представлений узлов

Содержание:

1. Введение

В данной работе рассматривается решение задачи классификации узлов с применением различных методов векторных представлений узлов. Классификация узлов графа является важным инструментом анализа и обработки графовых данных. Такая задача может встретиться в различных областях, например, в работе с социальными сетями или рекомендательными системами. Она состоит в присвоении меток узлам на графе на основе свойств узлов и взаимосвязей между ними.

Для успешного решения задачи классификации часто используются методы векторных представлений узлов, которые позволяют представить узлы в виде векторов, что может быть использовано при применении алгоритмов машинного обучения.

В исследовании будут применяться различные методы веторных представлений, а также несколько алгоритмов классификации узлов сети. По итогам экспериментального сравнения можно будет сделать выводы о том, какие методы и почему лучше решают поставленную задачу лучше или хуже.

В качестве данных для работы был выбран следующий набор:

Сеть из 100 тысяч пользователей, из которых около 5 тысяч были помечены как высказывающие ненависть пользователи или нет. Также для каждого пользователя были предоставлены несколько актрибутов, связанных с их активностью в социальной сети. Ребра графа являются ретвитами пользователей, поэтому представленный граф является направленным.

Таким образом, классификация будет заключаться в нахождении пользователей, разжигающих ненависть - бинарная классификация. Задача обнаружения пользователей, нарушающих правила сообщества, в наше время очень актуальна, поэтому данная работа имеет прикладное значение.

Задачи:

1. Получить набор данных, готовый к применению
2. Создать векторные представления узлов различными методами
3. Провести классификацию узлов
4. Сделать выводы о примененных методах

2. Основная часть

In [25]:
```
import networkx as nx
import pandas as pd
import numpy as np
from sklearn.decomposition import TruncatedSVD
from node2vec import Node2Vec
from karateclub.node_embedding.neighbourhood import GraRep, HOPE, DeepWalk
import random
from scipy.sparse import coo_matrix
from karateclub.node_embedding.attributed import SINE
import seaborn as sns
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.metrics import classification_report, f1_score
from sklearn import svm
from sklearn.ensemble import RandomForestClassifier
from sklearn.utils.class_weight import compute_class_weight
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from torch_geometric.nn import GCNConv
from torch_geometric.data import Data
```

2.1. Работа с данными

In [26]:
```
from google.colab import drive
drive.mount('/content/drive')
```

```
Drive already mounted at /content/drive; to attempt to forcibly remount, call dri
ve.mount("/content/drive", force_remount=True).
```

In [27]:
```
G0 = nx.read_edgelist('drive/MyDrive/users.edges', create_using = nx.DiGraph())
df0 = pd.read_csv('drive/MyDrive/users_neighborhood_anon.csv')
```

```
#df0.head()
```

Для классификации оставим только узлы, имеющие пометку.

In [28]:
```
df = df0[df0.hate != 'other'].reset_index(drop=True)
df
```

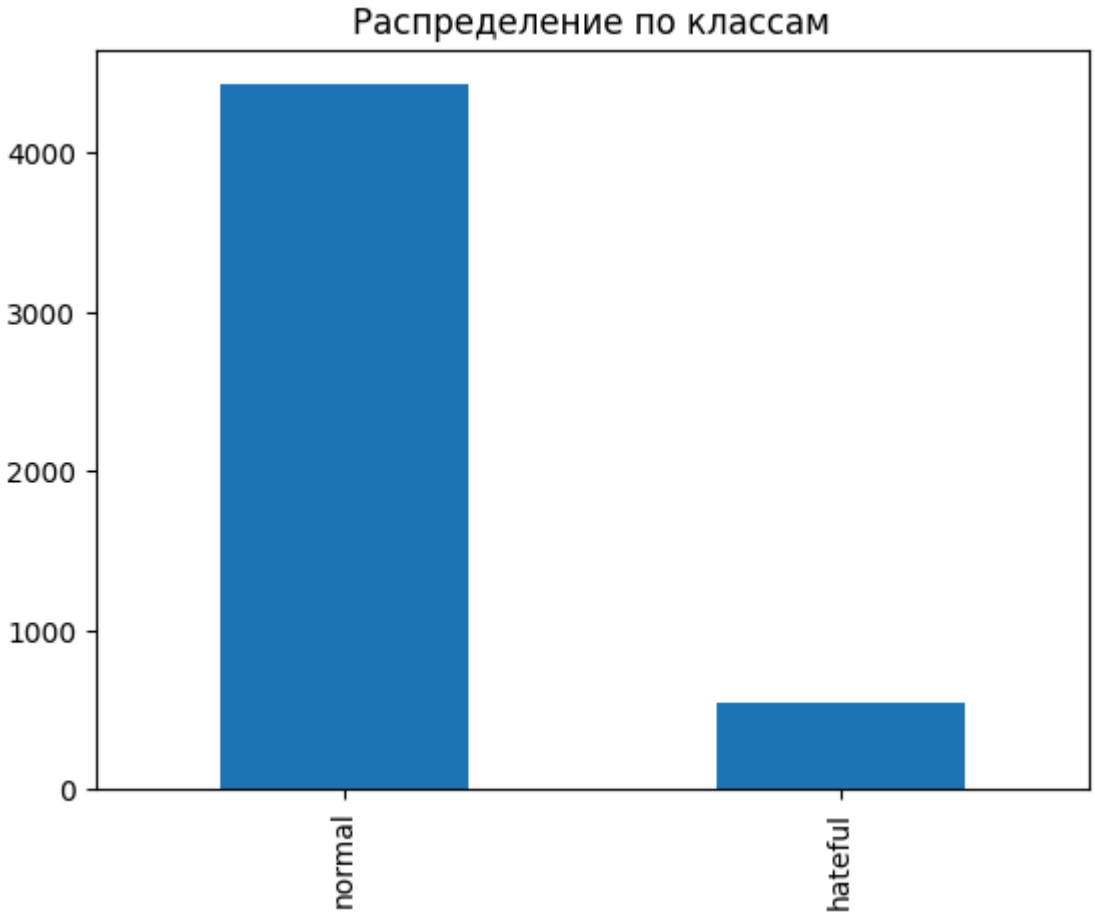
Out[28]:

	user_id	hate	hate_neigh	normal_neigh	statuses_count	followers_count	follo
0	0	normal	True	True	101767	3504	
1	22	normal	False	True	111948	1586	
2	29	normal	False	True	28222	27545	
3	44	normal	False	True	3101	2173	
4	85	normal	False	True	49169	2321	
...	...	...	...	...	...	...	
4966	100332	normal	False	True	104905	3700	
4967	100338	normal	False	True	5607	676	
4968	100362	normal	False	True	34	7534	
4969	100380	hateful	True	True	14608	49583	
4970	100385	normal	False	True	6868	433	

4971 rows × 1039 columns

In [8]:
```
print(df.hate.value_counts())
df.hate.value_counts().plot(kind='bar')
plt.title('Распределение по классам')
plt.show()
```

```
normal     4427
hateful     544
Name: hate, dtype: int64
```



Можно заметить, что классы являются несбалансированными, что может повлиять на дальнейшие результаты классификации, поэтому надо будет учесть это при выборе параметров моделей

In [30]:
```
df.hate = pd.get_dummies(df, columns=['hate'], drop_first = True)['hate_normal']
df
```

Out[30]:

	user_id	hate	hate_neigh	normal_neigh	statuses_count	followers_count	followe
0	0	1	True	True	101767	3504	
1	22	1	False	True	111948	1586	
2	29	1	False	True	28222	27545	
3	44	1	False	True	3101	2173	
4	85	1	False	True	49169	2321	
...	...	...	...	...	...	...	
4966	100332	1	False	True	104905	3700	
4967	100338	1	False	True	5607	676	
4968	100362	1	False	True	34	7534	
4969	100380	0	True	True	14608	49583	
4970	100385	1	False	True	6868	433	

4971 rows × 1039 columns

In [31]:
```
subgraph = G0.subgraph(map(str, df.user_id.to_list()))
print(subgraph)
```

```
DiGraph with 4971 nodes and 15141 edges
```

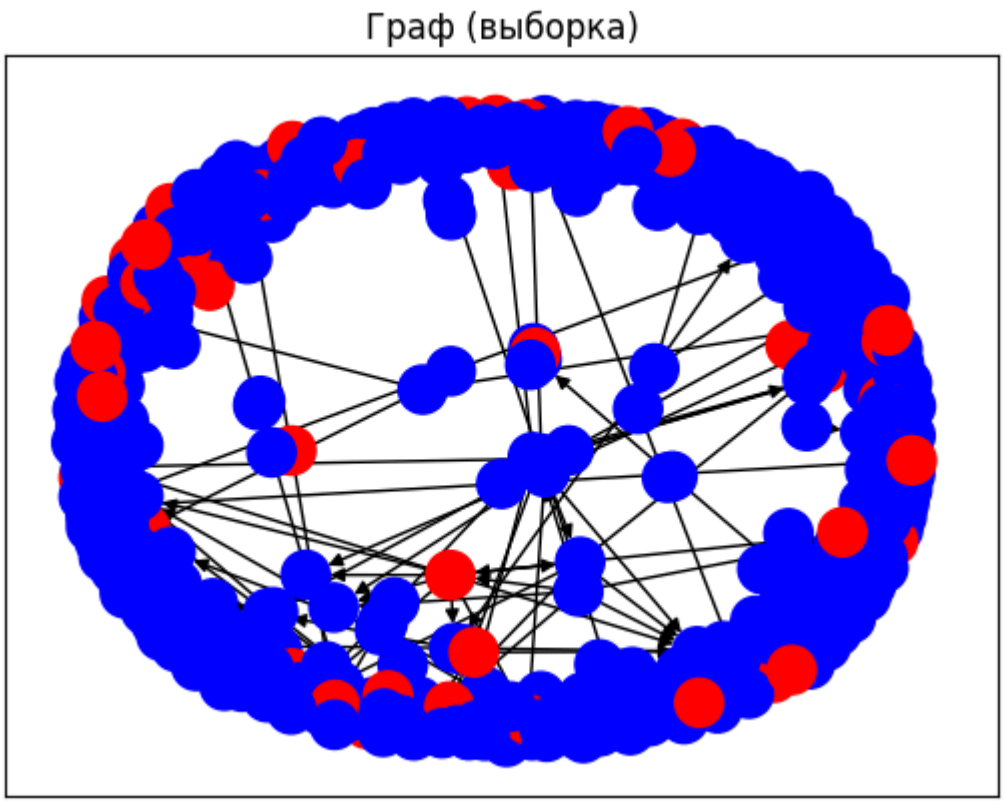
Удалим петли, так как они не несут в себе смысловой нагрузки в рамках выбранных данных

In [32]:
```
G = nx.relabel_nodes(subgraph, dict(zip(subgraph, map(int, subgraph.nodes()))))
G = nx.convert_node_labels_to_integers(G, ordering='sorted')
G.remove_edges_from(list(nx.selfloop_edges(G)))
print(G)
#nx.draw(G)
```

```
DiGraph with 4971 nodes and 10170 edges
```

Далее представлена десятая часть сети, где красным помечены пользователи, разжигающие ненависть

In [12]:
```
node_colors = list(map(lambda x: 'r' if x == 0 else 'b', df.hate))
nx.draw_networkx(G.subgraph(df.index.to_list()[::10]), node_color = node_colors[
plt.title('Граф (выборка)')
plt.show()
```



2.2. Векторные представления узлов

Рассмотрим некоторые методы векторных представлений узлов, которые основаны на:

1. Матричном разложении
2. Случайных блужданиях

Для удобства при применении каждого из методов будем получать вектора одинаковой размерности

2.2.1. Матричное разложение

Сингулярное разложение матрицы смежности узлов

Самой простой способ получения векторных представлений узлов из графа с использованием матричной факторизации заключается в использовании метода Singular Value Decomposition (SVD), который уменьшает размерность матрицы смежности узлов посредством сингулярного разложения

In [34]:
```
A = nx.to_numpy_array(G)
svd = TruncatedSVD(n_components=32)
vec_svd = svd.fit_transform(A)
vec_svd.shape
```

Out[34]:
```
(4971, 32)
```

GraRep (Learning Graph Representations with Global Structural Information)

Метод GraRep основывается на вычислении матрицы переходов, элементы который являются вероятностями перехода из одного узла в другой, рассчитанными на основе количества общих соседей этих узлов

In [35]:
```
model = GraRep(dimensions = 8, order = 4)
model.fit(G)
```

In [36]:
```
vec_grarep = model.get_embedding()
vec_grarep.shape
```

Out[36]: (4971, 32)

HOPE (Higher-Order Proximity Embeddings)

Метод HOPE основан на идее, что узлы, которые имеют сходную структуру связей с другими узлами, должны иметь схожие векторные представления. Особенность алгоритма в том, что он учитывает связи выше второго порядка

In [37]:
```
model = HOPE(dimensions = 32)
model.fit(G)
```

In [38]:
```
vec_hope = model.get_embedding()
vec_hope.shape
```

Out[38]: (4971, 32)

2.2.2. Случайные блуждания

Методы, который вычисляют векторные представления узла на основе случайных блужданий в графе: DeepWalk и Node2Vec.

Node2Vec - это вариация DeepWalk, которая вводит смещенные случайные блуждания. Случайными блужданиями управляют два параметра: p уменьшает вероятность повторного посещения предыдущего узла, в то время как q уменьшает вероятность перехода к узлам, которые не были соседями исходного узла.

DeepWalk

In [39]:
```
model = DeepWalk(dimensions = 32, walk_length=30, workers=4, walk_number = 50)
model.fit(G)
```

```
WARNING:gensim.models.word2vec:Both hierarchical softmax and negative sampling ar
e activated. This is probably a mistake. You should set either 'hs=0' or 'negativ
e=0' to disable one of them.
```

In [40]:
```
vec_deepwalk = model.get_embedding()
vec_deepwalk.shape
```

Out[40]: (4971, 32)

Node2Vec

In [41]:
```
node2vec = Node2Vec(G, dimensions=32, walk_length=50, num_walks=30, workers=4, p
model = node2vec.fit(window=10, min_count=1)
vec_node2vec = np.array([model.wv[node] for node in G.nodes()])
vec_node2vec.shape
```

```
Computing transition probabilities:   0%|          | 0/4971 [00:00<?, ?it/s]
```

Out[41]: (4971, 32)

SINE (Scalable Incomplete Network Embedding)

Метод SINE позволяет использовать атрибуты узлов для построения векторов.

Процедура неявно факторизует совместную мощность матрицы смежности и матрицы признаков. Декомпозиция выполняется на основе усеченных случайных блужданий, и мощности матрицы смежности объединяются.

In [42]:
```
df_features = df.iloc[:, 2:].select_dtypes(include='number')
features = df_features.to_numpy()
```

In [43]:
```
X = coo_matrix(features)
model = SINE(dimensions = 32)
model.fit(G, X)
```

In [44]:
```
vec_sine = model.get_embedding()
vec_sine.shape
```

Out[44]: (4971, 32)

2.3. Классификация узлов

Чтобы получить более точные результаты сравнения, классификацию будем проводить разными методами:

1. Методы машинного обучения
2. Нейронная сеть

2.3.1. Методы машинного обучения

Разделим данные на обучающую и тестовую выборки, а также найдем веса классов

In [45]:
```
train_mask, test_mask = train_test_split(df.index, test_size=0.2)
y = df.hate
class_weights = compute_class_weight('balanced', classes=np.unique(y), y=y)
cw = dict(zip(np.unique(y), class_weights))
```

SVC (Support Vector Classifier)

Работает путем нахождения гиперплоскости в многомерном пространстве, которая разделяет точки данных на разные классы.

In [46]:
```
clf = svm.SVC(class_weight = cw)
clf.fit(vec_svd[train_mask], y[train_mask])
```

Out[46]:
```
SVC(class_weight={0: 4.568933823529412, 1: 0.5614411565394172})
```

In [47]:
```
svd_res = clf.predict(vec_svd[test_mask])
svc_res = [f1_score(y[test_mask], svd_res, average = 'macro')]
print(classification_report(svd_res, y[test_mask]))
```

```
              precision    recall  f1-score   support

           0       0.04      0.10      0.05        41
           1       0.96      0.89      0.92       954

    accuracy                           0.85       995
   macro avg       0.50      0.49      0.49       995
weighted avg       0.92      0.85      0.89       995
```

In [48]:
```
clf = svm.SVC(class_weight = cw)
clf.fit(vec_grarep[train_mask], y[train_mask])
```

Out[48]:
```
SVC(class_weight={0: 4.568933823529412, 1: 0.5614411565394172})
```

In [49]:
```
grarep_res = clf.predict(vec_grarep[test_mask])
svc_res.append(f1_score(y[test_mask], grarep_res, average = 'macro'))
print(classification_report(grarep_res, y[test_mask]))
```

```
              precision    recall  f1-score   support

           0       0.70      0.53      0.60       147
           1       0.92      0.96      0.94       848

    accuracy                           0.90       995
   macro avg       0.81      0.75      0.77       995
weighted avg       0.89      0.90      0.89       995
```

In [50]:
```
clf = svm.SVC(class_weight = cw)
clf.fit(vec_hope[train_mask], y[train_mask])
```

Out[50]:
```
SVC(class_weight={0: 4.568933823529412, 1: 0.5614411565394172})
```

In [51]:
```
hope_res = clf.predict(vec_hope[test_mask])
svc_res.append(f1_score(y[test_mask], hope_res, average = 'macro'))
print(classification_report(hope_res, y[test_mask]))
```

```
                precision    recall  f1-score   support

           0       0.64      0.60      0.62       120
           1       0.95      0.95      0.95       875

    accuracy                           0.91       995
   macro avg       0.79      0.78      0.79       995
weighted avg       0.91      0.91      0.91       995
```

In [52]:
```
clf = svm.SVC(class_weight = cw)
clf.fit(vec_deepwalk[train_mask], y[train_mask])
```

Out[52]:
```
SVC(class_weight={0: 4.568933823529412, 1: 0.5614411565394172})
```

In [53]:
```
deepwalk_res = clf.predict(vec_deepwalk[test_mask])
svc_res.append(f1_score(y[test_mask], deepwalk_res, average = 'macro'))
print(classification_report(deepwalk_res, y[test_mask]))
```

```
                precision    recall  f1-score   support

           0       0.64      0.60      0.62       121
           1       0.94      0.95      0.95       874

    accuracy                           0.91       995
   macro avg       0.79      0.77      0.78       995
weighted avg       0.91      0.91      0.91       995
```

In [54]:
```
clf = svm.SVC(class_weight = cw)
clf.fit(vec_node2vec[train_mask], y[train_mask])
```

Out[54]:
```
SVC(class_weight={0: 4.568933823529412, 1: 0.5614411565394172})
```

In [55]:
```
node2vec_res = clf.predict(vec_node2vec[test_mask])
svc_res.append(f1_score(y[test_mask], node2vec_res, average = 'macro'))
print(classification_report(node2vec_res, y[test_mask]))
```

```
                precision    recall  f1-score   support

           0       0.22      0.10      0.14       254
           1       0.74      0.88      0.81       741

    accuracy                           0.68       995
   macro avg       0.48      0.49      0.47       995
weighted avg       0.61      0.68      0.63       995
```

In [56]:
```
clf = svm.SVC(class_weight = cw)
clf.fit(vec_sine[train_mask], y[train_mask])
```

Out[56]:

SVC

```
SVC(class_weight={0: 4.568933823529412, 1: 0.5614411565394172})
```

In [57]:

```
sine_res = clf.predict(vec_sine[test_mask])
svc_res.append(f1_score(y[test_mask], sine_res, average = 'macro'))
print(classification_report(sine_res, y[test_mask]))
```

```
              precision    recall  f1-score   support

           0       0.68      0.59      0.63       129
           1       0.94      0.96      0.95       866

    accuracy                           0.91       995
   macro avg       0.81      0.77      0.79       995
weighted avg       0.91      0.91      0.91       995
```

In [58]:

```
pd.DataFrame(svc_res, columns=['f1_score'], index = ['svd','grarep','hope','deep
```

Out[58]:

	f1_score
sine	0.789910
hope	0.785316
deepwalk	0.783686
grarep	0.771407
svd	0.486677
node2vec	0.471015

Лучший результат по f1-score показал метод SINE, который при обучении векторов использует атрибуты узлов. Атрибуты узлов в свою очередь являются своеобразными характеристика поведения пользователей, поэтому метод может быть очень показателен при наличии таких данных.

Второе место у HOPE. Можно предположить, что это происходит из-за того, что HOPE, в отличие от остальных методов, рассматривает связи выше второго порядка, то есть углубляется в сеть и взаимосвязи между пользователями. В рамках взятых данных - социальная сеть с ретвитами - это может означать, что пользователи, разжигающие ненависть, имеют определенные не сразу заметные сходства.

На последних местах svd и Node2Vec. Первое не вызывает особых вопросов, в связи с поверхностностью построения векторов. Node2Vec, вероятно, показывает плохой результат из-за неправильно подобрабных параметров при обучении векторов.

Random Forest

Работает путем построения множества деревьев решений и объединения их предсказаний.

In [59]:
```
clf = RandomForestClassifier(class_weight = cw)
clf.fit(vec_svd[train_mask], y[train_mask])
```

Out[59]:
```
RandomForestClassifier(class_weight={0: 4.568933823529412,
                                     1: 0.5614411565394172})
```

In [60]:
```
svd_res = clf.predict(vec_svd[test_mask])
rf_res = [f1_score(y[test_mask], svd_res, average = 'macro')]
print(classification_report(svd_res, y[test_mask]))
```

```
              precision    recall  f1-score   support

           0       0.01      0.10      0.02        10
           1       0.99      0.89      0.94       985

    accuracy                           0.88       995
   macro avg       0.50      0.49      0.48       995
weighted avg       0.98      0.88      0.93       995
```

In [61]:
```
clf = RandomForestClassifier(class_weight = cw)
clf.fit(vec_grarep[train_mask], y[train_mask])
```

Out[61]:
```
RandomForestClassifier(class_weight={0: 4.568933823529412,
                                     1: 0.5614411565394172})
```

In [62]:
```
grarep_res = clf.predict(vec_grarep[test_mask])
rf_res.append(f1_score(y[test_mask], grarep_res, average = 'macro'))
print(classification_report(grarep_res, y[test_mask]))
```

```
              precision    recall  f1-score   support

           0       0.46      0.64      0.54        81
           1       0.97      0.93      0.95       914

    accuracy                           0.91       995
   macro avg       0.72      0.79      0.74       995
weighted avg       0.93      0.91      0.92       995
```

In [63]:
```
clf = RandomForestClassifier(class_weight = cw)
clf.fit(vec_hope[train_mask], y[train_mask])
```

Out[63]:
```
RandomForestClassifier(class_weight={0: 4.568933823529412,
                                     1: 0.5614411565394172})
```

In [64]:
```
hope_res = clf.predict(vec_hope[test_mask])
rf_res.append(f1_score(y[test_mask], hope_res, average = 'macro'))
print(classification_report(hope_res, y[test_mask]))
```

```
              precision    recall  f1-score   support

           0       0.39      0.76      0.52        58
           1       0.98      0.93      0.95       937

    accuracy                           0.92       995
   macro avg       0.69      0.84      0.74       995
weighted avg       0.95      0.92      0.93       995
```

In [65]:
```
clf = RandomForestClassifier(class_weight = cw)
clf.fit(vec_deepwalk[train_mask], y[train_mask])
```

Out[65]:
```
RandomForestClassifier(class_weight={0: 4.568933823529412,
                                     1: 0.5614411565394172})
```

In [66]:
```
deepwalk_res = clf.predict(vec_deepwalk[test_mask])
rf_res.append(f1_score(y[test_mask], deepwalk_res, average = 'macro'))
print(classification_report(deepwalk_res, y[test_mask]))
```

```
              precision    recall  f1-score   support

           0       0.16      0.86      0.27        21
           1       1.00      0.90      0.95       974

    accuracy                           0.90       995
   macro avg       0.58      0.88      0.61       995
weighted avg       0.98      0.90      0.93       995
```

In [67]:
```
clf = RandomForestClassifier(class_weight = cw)
clf.fit(vec_node2vec[train_mask], y[train_mask])
```

Out[67]:
```
RandomForestClassifier(class_weight={0: 4.568933823529412,
                                     1: 0.5614411565394172})
```

In [68]:
```
node2vec_res = clf.predict(vec_node2vec[test_mask])
rf_res.append(f1_score(y[test_mask], node2vec_res, average = 'macro'))
print(classification_report(node2vec_res, y[test_mask]))
```

```
              precision    recall  f1-score   support

           0       0.00      0.00      0.00         1
           1       1.00      0.89      0.94       994

    accuracy                           0.89       995
   macro avg       0.50      0.44      0.47       995
weighted avg       1.00      0.89      0.94       995
```

In [69]:
```
clf = RandomForestClassifier(class_weight = cw)
clf.fit(vec_sine[train_mask], y[train_mask])
```

Out[69]:

```
RandomForestClassifier(class_weight={0: 4.568933823529412,
                                     1: 0.5614411565394172})
```

In [70]:

```
sine_res = clf.predict(vec_sine[test_mask])
rf_res.append(f1_score(y[test_mask], sine_res, average = 'macro'))
print(classification_report(sine_res, y[test_mask]))
```

```
              precision    recall  f1-score   support

           0       0.42      0.85      0.56        55
           1       0.99      0.93      0.96       940

    accuracy                           0.93       995
   macro avg       0.71      0.89      0.76       995
weighted avg       0.96      0.93      0.94       995
```

In [71]:

```
pd.DataFrame(rf_res, columns=['f1_score'], index = ['svd','grarep','hope','deepw
```

Out[71]:

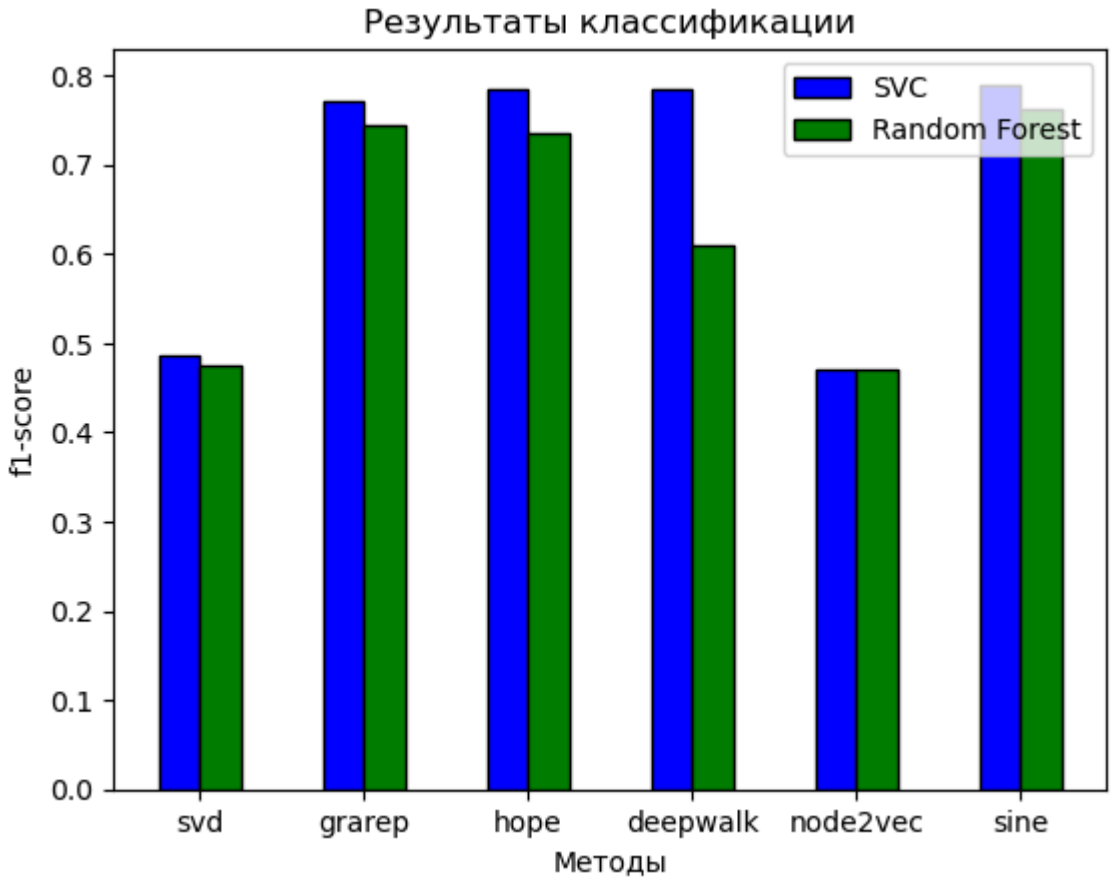
	f1_score
sine	0.761415
grarep	0.744667
hope	0.736296
deepwalk	0.609221
svd	0.476077
node2vec	0.469899

В случае модели Случайного леса лучшим стал снова метод SINE.

Можно еще выделить метод GraRep, который занял 2 место по f1-score, несмотря на относительно небольшую вычислительную мощность. Данный метод учитывает первостепенные и второстепенные связи, на основе которых, как показала практика, можно получить хорошие результаты классификации.

In [75]:

```
r = np.arange(6)
width = 0.25

plt.bar(r, svc_res, color = 'b', width = width, edgecolor = 'black', label='SVC'
plt.bar(r + width, rf_res, color = 'g', width = width, edgecolor = 'black', labe

plt.xlabel("Методы")
plt.ylabel("f1-score")
plt.title("Результаты классификации")

plt.xticks(r + width/2,['svd','grarep','hope','deepwalk', 'node2vec', 'sine'])
plt.legend()
plt.show()
```



Обобщая результаты проведенной классификации двумя алгоритмами машинного обучения, можено сделать вывод, что для социальной сети лучше подходит SVC модель, которая находит оптимальную гиперплоскость для разделения классов. Так как данные имеют векторное представление, такой результат можно назвать закономерным

Интересно, что DeepWalk, который использует только случайные блуждания для построения векторных представлений, имеет большое различие между f1-score

2.3.2. Нейронная сеть

Далее попробуем классифицировать узлы при помощи графовой сверточной сети.

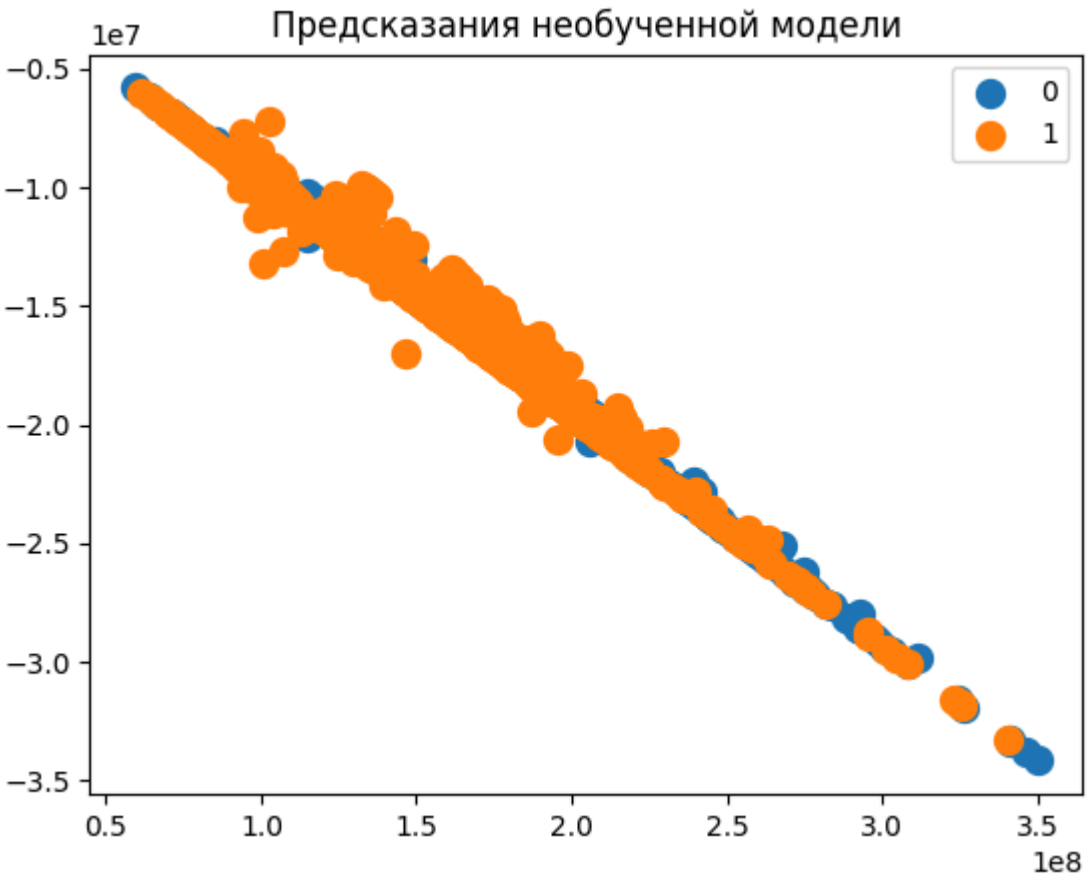
Сначала в качестве признаков узлов будем использовать атрибуты узлов, уменьшенные в размерности с помощью сингулярного разложения, а после - векторные представления узлов, полученные ранее

In [93]:
```
class GCN(nn.Module):
    def __init__(self, n_input, n_hidden, n_output):
        super().__init__()
        self.conv1 = GCNConv(n_input, n_hidden)
        self.conv2 = GCNConv(n_hidden, n_output)

    def forward(self, data):
        x, edge_index = data.x, data.edge_index
        x = F.relu(self.conv1(x, edge_index))
        x = self.conv2(x, edge_index)
        return x
```

In [94]:
```
n_epochs = 301
n_input = 32
n_hidden = 128
n_out = 2
```

In [95]:
```
def visualize():
    model.eval()
    with torch.no_grad():
        predictions = model(data)

    for cl in labels.unique():
        plt.scatter(predictions[cl == labels, 0], predictions[cl == labels, 1],
    plt.legend()
    plt.show()
```

In [75]:
```
model = GCN(n_input, n_hidden, n_out)

svd = TruncatedSVD(n_components=32)
features32 = svd.fit_transform(df_features.fillna(df.mean(numeric_only=True)).to

labels = torch.tensor(y).to(torch.int64)
edges = torch.tensor(list(G.edges)).t().contiguous().long()
data = Data(x = torch.from_numpy(features32).to(torch.float32), edge_index = edg

plt.title('Предсказания необученной модели')
visualize()
```

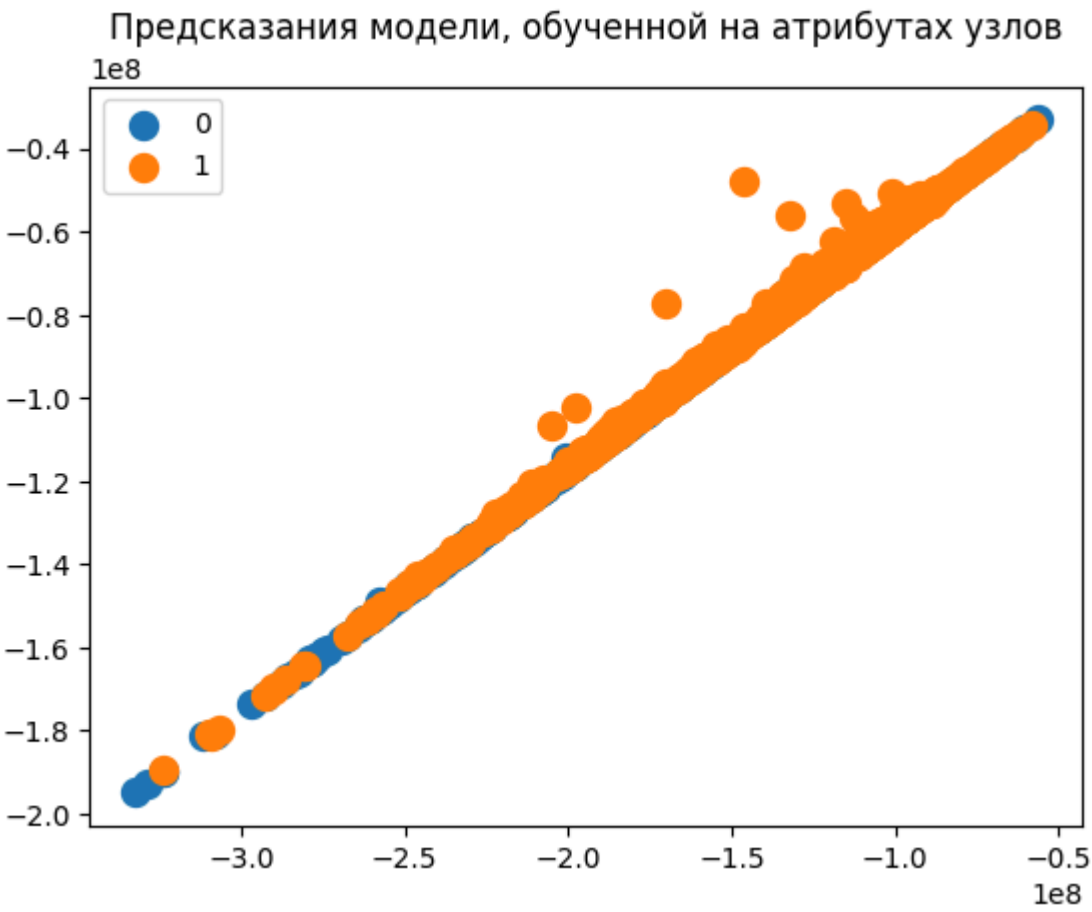


In [97]:
```
model = GCN(n_input, n_hidden, n_out)

optimizer = optim.Adam(model.parameters(), lr=.01)
criterion = nn.CrossEntropyLoss()
```

```
for epoch in range(n_epochs):
    logits = model(data)
    loss = criterion(logits[train_mask], labels[train_mask])

    loss.backward()
    optimizer.step()
    optimizer.zero_grad()

    with torch.no_grad():
        predictions = logits.argmax(dim=1)
        train_acc = (predictions[train_mask] == labels[train_mask]).float().mean
        test_acc = (predictions[test_mask] == labels[test_mask]).float().mean()

    if not epoch % 20:
        print(f'In epoch {epoch}, train acc: {train_acc:.3f}, test acc: {test_ac

gcn_res = [f1_score(predictions[test_mask], labels[test_mask], average = 'macro'
```

```
In epoch 0, train acc: 0.109, test acc: 0.113
In epoch 20, train acc: 0.891, test acc: 0.887
In epoch 40, train acc: 0.891, test acc: 0.887
In epoch 60, train acc: 0.891, test acc: 0.887
In epoch 80, train acc: 0.891, test acc: 0.887
In epoch 100, train acc: 0.891, test acc: 0.887
In epoch 120, train acc: 0.891, test acc: 0.887
In epoch 140, train acc: 0.134, test acc: 0.128
In epoch 160, train acc: 0.891, test acc: 0.887
In epoch 180, train acc: 0.891, test acc: 0.887
In epoch 200, train acc: 0.891, test acc: 0.887
In epoch 220, train acc: 0.891, test acc: 0.887
In epoch 240, train acc: 0.891, test acc: 0.887
In epoch 260, train acc: 0.891, test acc: 0.887
In epoch 280, train acc: 0.891, test acc: 0.887
In epoch 300, train acc: 0.891, test acc: 0.887
```

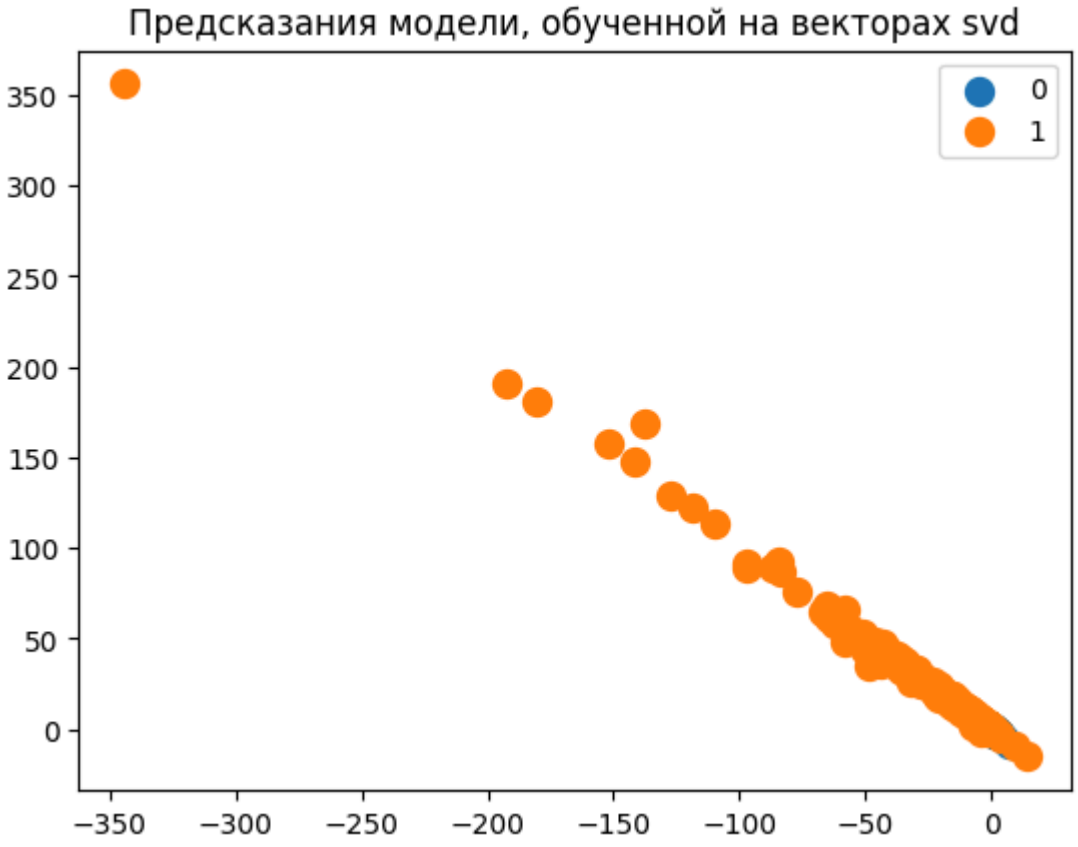
In [77]:
```
plt.title('Предсказания модели, обученной на атрибутах узлов')
visualize()
```



In [99]:
```
model = GCN(n_input, n_hidden, n_out)

optimizer = optim.Adam(model.parameters(), lr=.01)
criterion = nn.CrossEntropyLoss()

data = Data(x = torch.from_numpy(vec_svd).to(torch.float32), edge_index = edges)

for epoch in range(n_epochs):
    logits = model(data)
    loss = criterion(logits[train_mask], labels[train_mask])

    loss.backward()
    optimizer.step()
    optimizer.zero_grad()

    with torch.no_grad():
        predictions = logits.argmax(dim=1)
        train_acc = (predictions[train_mask] == labels[train_mask]).float().mean
        test_acc = (predictions[test_mask] == labels[test_mask]).float().mean()

    if not epoch % 40:
        print(f'In epoch {epoch}, train acc: {train_acc:.3f}, test acc: {test_ac

gcn_res.append(f1_score(predictions[test_mask], labels[test_mask], average = 'ma
```

```
In epoch 0, train acc: 0.407, test acc: 0.407
In epoch 40, train acc: 0.892, test acc: 0.884
In epoch 80, train acc: 0.892, test acc: 0.874
In epoch 120, train acc: 0.910, test acc: 0.896
In epoch 160, train acc: 0.917, test acc: 0.904
In epoch 200, train acc: 0.922, test acc: 0.906
In epoch 240, train acc: 0.924, test acc: 0.909
In epoch 280, train acc: 0.926, test acc: 0.910
```

In [79]:
```
plt.title('Предсказания модели, обученной на векторах svd')
visualize()
```

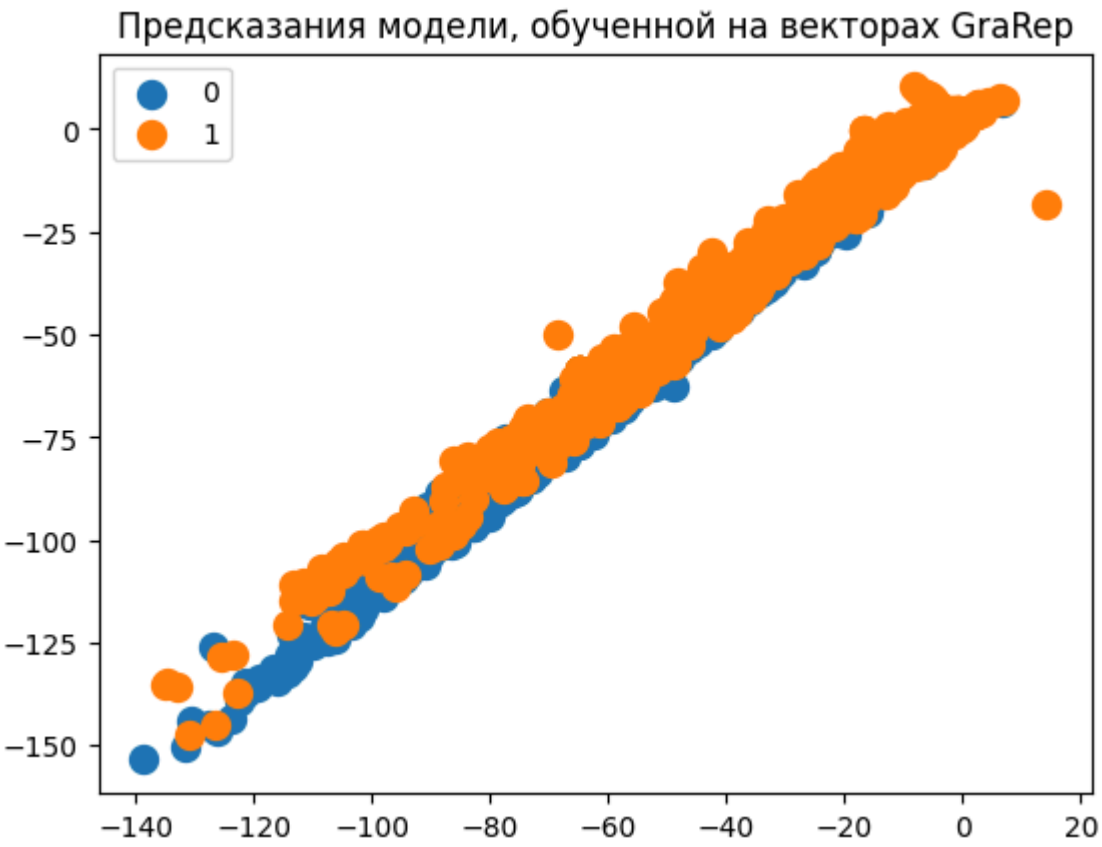


In [101…
```
model = GCN(n_input, n_hidden, n_out)

optimizer = optim.Adam(model.parameters(), lr=.01)
criterion = nn.CrossEntropyLoss()

data = Data(x = torch.from_numpy(vec_grarep).to(torch.float32), edge_index = edg

for epoch in range(n_epochs):
    logits = model(data)
    loss = criterion(logits[train_mask], labels[train_mask])

    loss.backward()
    optimizer.step()
    optimizer.zero_grad()

    with torch.no_grad():
        predictions = logits.argmax(dim=1)
        train_acc = (predictions[train_mask] == labels[train_mask]).float().mean
        test_acc = (predictions[test_mask] == labels[test_mask]).float().mean()

    if not epoch % 40:
        print(f'In epoch {epoch}, train acc: {train_acc:.3f}, test acc: {test_ac
```

```
gcn_res.append(f1_score(predictions[test_mask], labels[test_mask], average = 'ma
```

```
In epoch 0, train acc: 0.138, test acc: 0.144
In epoch 40, train acc: 0.894, test acc: 0.883
In epoch 80, train acc: 0.907, test acc: 0.908
In epoch 120, train acc: 0.906, test acc: 0.904
In epoch 160, train acc: 0.925, test acc: 0.922
In epoch 200, train acc: 0.907, test acc: 0.902
In epoch 240, train acc: 0.922, test acc: 0.911
In epoch 280, train acc: 0.923, test acc: 0.914
```

In [81]:
```
plt.title('Предсказания модели, обученной на векторах GraRep')
visualize()
```

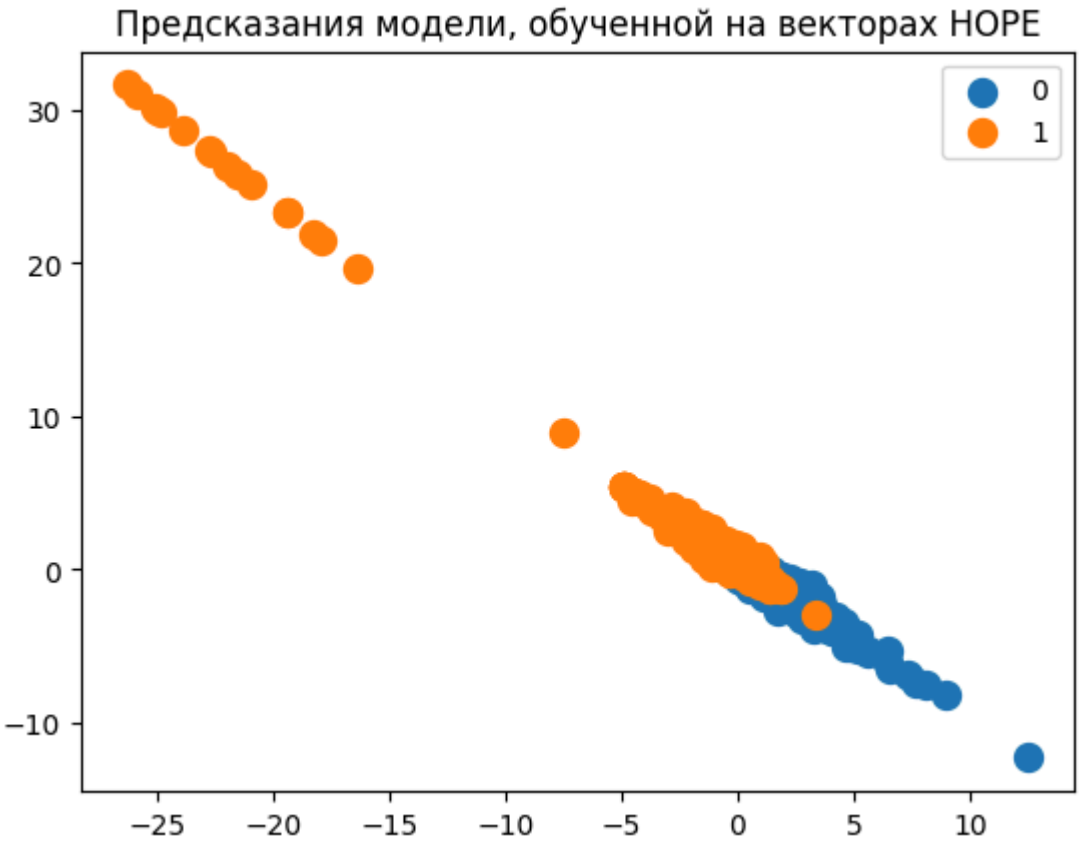


In [103…
```
model = GCN(n_input, n_hidden, n_out)

optimizer = optim.Adam(model.parameters(), lr=.01)
criterion = nn.CrossEntropyLoss()

data = Data(x = torch.from_numpy(vec_hope).to(torch.float32), edge_index = edges

for epoch in range(n_epochs):
    logits = model(data)
    loss = criterion(logits[train_mask], labels[train_mask])

    loss.backward()
    optimizer.step()
    optimizer.zero_grad()

    with torch.no_grad():
        predictions = logits.argmax(dim=1)
        train_acc = (predictions[train_mask] == labels[train_mask]).float().mean
        test_acc = (predictions[test_mask] == labels[test_mask]).float().mean()
```

```
    if not epoch % 40:
        print(f'In epoch {epoch}, train acc: {train_acc:.3f}, test acc: {test_ac

gcn_res.append(f1_score(predictions[test_mask], labels[test_mask], average = 'ma
```

```
In epoch 0, train acc: 0.524, test acc: 0.532
In epoch 40, train acc: 0.926, test acc: 0.921
In epoch 80, train acc: 0.932, test acc: 0.919
In epoch 120, train acc: 0.936, test acc: 0.922
In epoch 160, train acc: 0.943, test acc: 0.920
In epoch 200, train acc: 0.947, test acc: 0.918
In epoch 240, train acc: 0.949, test acc: 0.919
In epoch 280, train acc: 0.951, test acc: 0.916
```

In [83]:
```
plt.title('Предсказания модели, обученной на векторах HOPE')
visualize()
```

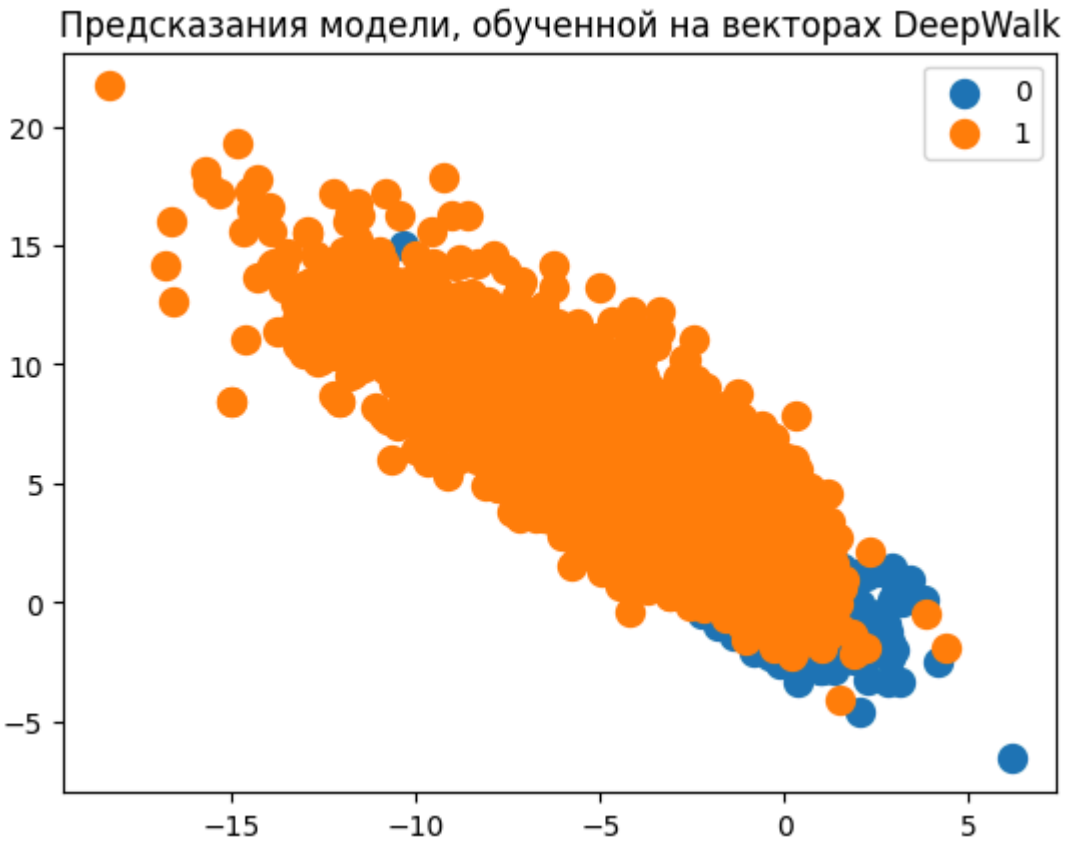


In [105…
```
model = GCN(n_input, n_hidden, n_out)

optimizer = optim.Adam(model.parameters(), lr=.01)
criterion = nn.CrossEntropyLoss()

data = Data(x = torch.from_numpy(vec_deepwalk).to(torch.float32), edge_index = e

for epoch in range(n_epochs):
    logits = model(data)
    loss = criterion(logits[train_mask], labels[train_mask])

    loss.backward()
    optimizer.step()
    optimizer.zero_grad()

    with torch.no_grad():
```

```
        predictions = logits.argmax(dim=1)
        train_acc = (predictions[train_mask] == labels[train_mask]).float().mean
        test_acc = (predictions[test_mask] == labels[test_mask]).float().mean()

    if not epoch % 40:
        print(f'In epoch {epoch}, train acc: {train_acc:.3f}, test acc: {test_ac

gcn_res.append(f1_score(predictions[test_mask], labels[test_mask], average = 'ma
```

```
In epoch 0, train acc: 0.818, test acc: 0.820
In epoch 40, train acc: 0.929, test acc: 0.924
In epoch 80, train acc: 0.935, test acc: 0.919
In epoch 120, train acc: 0.949, test acc: 0.910
In epoch 160, train acc: 0.960, test acc: 0.896
In epoch 200, train acc: 0.968, test acc: 0.886
In epoch 240, train acc: 0.972, test acc: 0.891
In epoch 280, train acc: 0.981, test acc: 0.892
```

In [85]:
```
plt.title('Предсказания модели, обученной на векторах DeepWalk')
visualize()
```

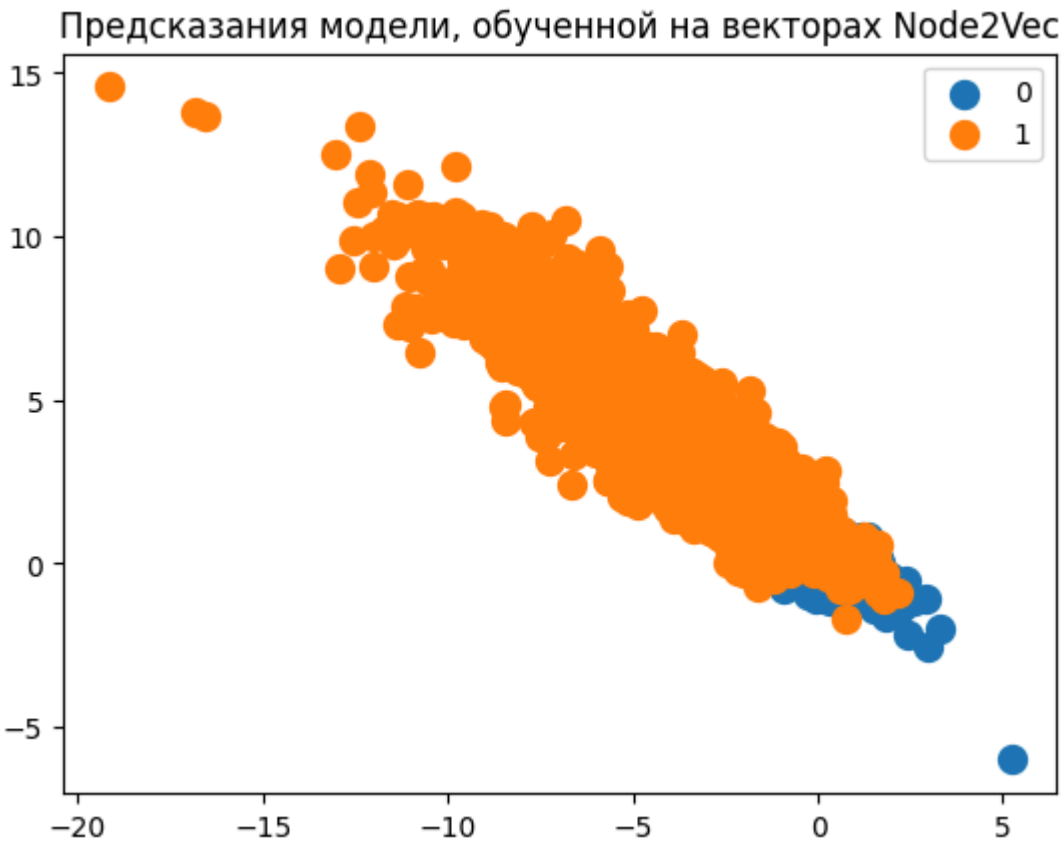


In [107…
```
model = GCN(n_input, n_hidden, n_out)

optimizer = optim.Adam(model.parameters(), lr=.01)
criterion = nn.CrossEntropyLoss()

data = Data(x = torch.from_numpy(vec_node2vec).to(torch.float32), edge_index = e

for epoch in range(n_epochs):
    logits = model(data)
    loss = criterion(logits[train_mask], labels[train_mask])

    loss.backward()
    optimizer.step()
```

```
    optimizer.zero_grad()

    with torch.no_grad():
        predictions = logits.argmax(dim=1)
        train_acc = (predictions[train_mask] == labels[train_mask]).float().mean
        test_acc = (predictions[test_mask] == labels[test_mask]).float().mean()

    if not epoch % 40:
        print(f'In epoch {epoch}, train acc: {train_acc:.3f}, test acc: {test_ac

gcn_res.append(f1_score(predictions[test_mask], labels[test_mask], average = 'ma
```

```
In epoch 0, train acc: 0.644, test acc: 0.636
In epoch 40, train acc: 0.895, test acc: 0.887
In epoch 80, train acc: 0.917, test acc: 0.902
In epoch 120, train acc: 0.926, test acc: 0.902
In epoch 160, train acc: 0.938, test acc: 0.893
In epoch 200, train acc: 0.943, test acc: 0.890
In epoch 240, train acc: 0.951, test acc: 0.883
In epoch 280, train acc: 0.953, test acc: 0.882
```

In [87]:
```
plt.title('Предсказания модели, обученной на векторах Node2Vec')
visualize()
```

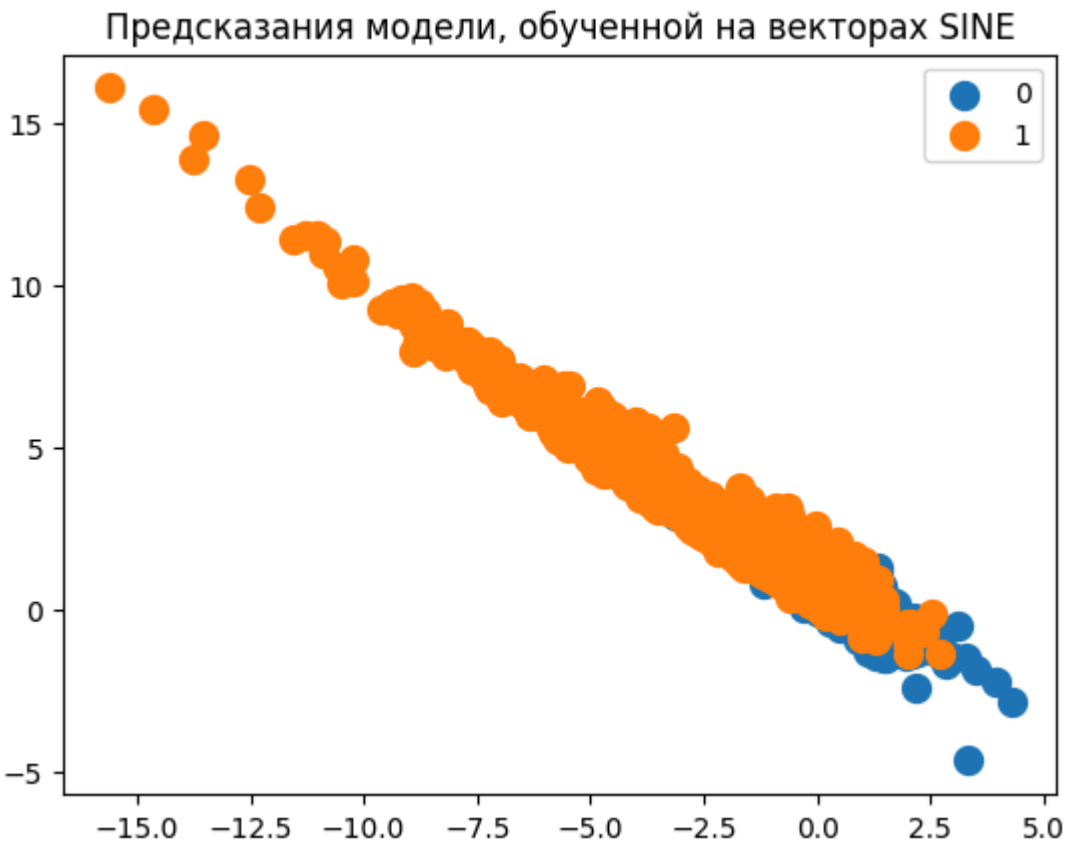


In [109…
```
model = GCN(n_input, n_hidden, n_out)

optimizer = optim.Adam(model.parameters(), lr=.01)
criterion = nn.CrossEntropyLoss()

data = Data(x = torch.from_numpy(vec_sine).to(torch.float32), edge_index = edges

for epoch in range(n_epochs):
    logits = model(data)
    loss = criterion(logits[train_mask], labels[train_mask])
```

```
    loss.backward()
    optimizer.step()
    optimizer.zero_grad()

    with torch.no_grad():
        predictions = logits.argmax(dim=1)
        train_acc = (predictions[train_mask] == labels[train_mask]).float().mean
        test_acc = (predictions[test_mask] == labels[test_mask]).float().mean()

    if not epoch % 40:
        print(f'In epoch {epoch}, train acc: {train_acc:.3f}, test acc: {test_ac

gcn_res.append(f1_score(predictions[test_mask], labels[test_mask], average = 'ma
```

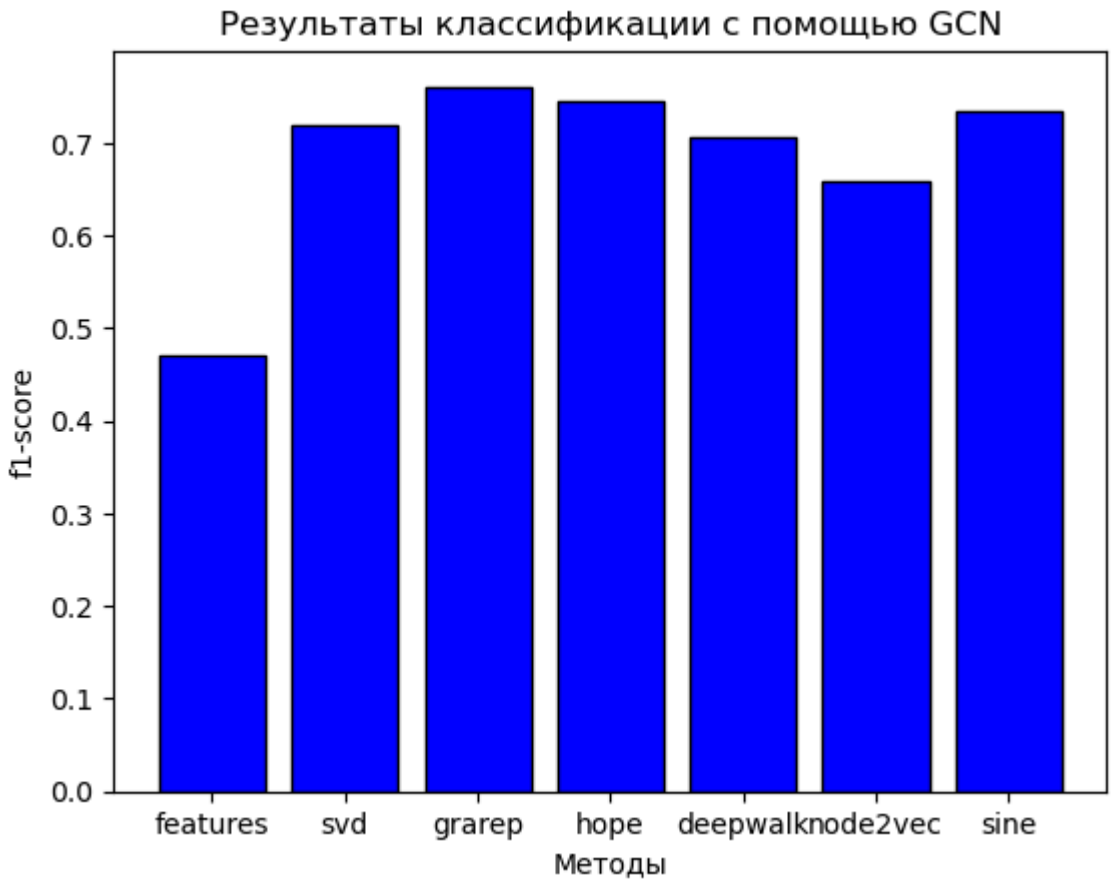
```
In epoch 0, train acc: 0.222, test acc: 0.225
In epoch 40, train acc: 0.922, test acc: 0.921
In epoch 80, train acc: 0.927, test acc: 0.917
In epoch 120, train acc: 0.930, test acc: 0.922
In epoch 160, train acc: 0.933, test acc: 0.920
In epoch 200, train acc: 0.938, test acc: 0.915
In epoch 240, train acc: 0.942, test acc: 0.911
In epoch 280, train acc: 0.945, test acc: 0.912
```

In [89]:
```
plt.title('Предсказания модели, обученной на векторах SINE')
visualize()
```



In [92]:
```
r = np.arange(7)

plt.bar(r, gcn_res, color = 'b', edgecolor = 'black')

plt.xlabel("Методы")
plt.ylabel("f1-score")
plt.title("Результаты классификации с помощью GCN")
```

```
plt.xticks(r,['features', 'svd','grarep','hope','deepwalk', 'node2vec', 'sine'])
plt.show()
```



In [112…
```
pd.DataFrame(gcn_res, columns=['f1_score'], index = ['features','svd','grarep','
```

Out[112…

	f1_score
grarep	0.760978
hope	0.746001
sine	0.733996
svd	0.719984
deepwalk	0.706149
node2vec	0.657994
features	0.470181

При одинаковых параметров обучения, лучше всего себя показали те же методы, что и раньше. При этом обучение только на атрибутах узлов дало довольно плохой результат, в отличие от векторных представлений, полученных в ходе работы.

3. Заключение

По итогам работы мы получили результаты классификации узлов сети несколькими способами. Можно сказать, что методы, глубоко рассматривающие взаимосвязи

между узлами, лучше справляются с задачей классификации. Конечно, данные выводы распространяются только на выбранный набор данных и, вероятнее всего, могут быть распространены только на данные со схожей структурой.

В рамках задачи классификации пользователей, разжигающих ненависть, хорошо себя показал метод SINE, учитывающий и отношения между объектами сети, и атрибуты узлов. А также методы HOPE и GraRep, первый их которых включает в рассмотрение связи выше второго порядка, а второй - первого и второго.

Говоря о алгоритмах классификации, поскольку для обучения не подбирались специальные параметры, сложно сделать вывод о том, какой является наилучшим в широком смысле. Однако в ходе проведенного эксперимента при прочих равных условиях наибольшие значения получились при использовании SVC.

Список используемых источников:

1. Данные - https://www.kaggle.com/datasets/manoelribeiro/hateful-users-on-twitter
2. Karate Club: An API Oriented Open-source Python Framework for Unsupervised Learning on Graphs (CIKM 2020) - https://github.com/benedekrozemberczki/karateclub
3. Под капотом графовых сетей - https://habr.com/ru/articles/794558/
4. Characterizing and Detecting Hateful Users on Twitter - https://www.researchgate.net/publication/365061339