Министерство науки и высшего образования Российской Федерации
Санкт-Петербургский Политехнический Университет Петра Великого
—
Институт компьютерных наук и кибербезопасности
Высшая школа технологий искусственного интеллекта

# Курсовая работа

по дисциплине «Машинное обучение. Часть 1»

Выполнил: студент группы
5140201/30301 С.П. Хомец
*<подпись>*

Проверил: Л.В. Уткин
д.т.н., профессор
*<подпись>*

Санкт-Петербург
2024 г.

# 1. Введение

## 1.1. Использованный датасет

В качестве датасет был выбран Parkinsons Disease Data Set. Он содержит описание набора данных о биомедицинских измерениях голоса для диагностики болезни Паркинсона. Данные включают биомедицинские измерения голоса 31 человека, из которых 23 имеют болезнь Паркинсона. Цель данных – различать здоровых людей от тех, у которых есть болезнь Паркинсона, согласно колонке “status”, где 0 – здоров, 1 – болезнь Паркинсона.

Информация об атрибутах:

- ‘name’ – имя субъекта и номер записи;
- ‘MDVP:Fo(Hz)’, ‘MDVP:Fhi(Hz)’, ‘MDVP:Flo(Hz)’ – различные характеристики частоты голоса;
- ‘MDVP:Jitter(%)’, ‘MDVP:Jitter(Abs)’, ‘MDVP:RAP’, ‘MDVP:PPQ’, ‘Jitter:DDP’ – различные меры вариации в фундаментальной частоте;
- ‘MDVP:Shimmer’, ‘MDVP:Shimmer(dB)’, ‘Shimmer:APQ3’, ‘Shimmer:APQ5’, ‘MDVP:APQ’, ‘Shimmer:DDA’ – различные меры вариации в амплитуде;
- ‘NHR’, ‘HNR’ – две меры соотношения шума к тональным компонентам в голосе;
- ‘status’ – здоровье субъекта;
- ‘RPDE’, ‘D2’ – две нелинейные меры динамической сложности;
- ‘DFA’ – экспонента фрактального масштабирования сигнала;
- ‘spread1’, ‘spread2’, ‘PPE’ – три нелинейные меры вариации фундаментальной частоты.

Сам набор данных небольшой, содержит 195 примеров, 22 признака.

## 1.2. Использованные модели

### 1.2.1. SVM

Имеется m-мерное пространство $R^m$. Каждый пример представлен точкой $x_i$ в нашем случае требуется бинарная классификация, поэтому каждому $x_i$ соответствует метка $y_i = \{-1, 1\}$.

Требуется построить разделяющую гиперплоскость:

$$g(x) = w^T x + b,$$

Такую, чтобы точки, лежащие по разные стороны от нее, имели разные метки:

$$w^T x + b > 0 => y = 1$$

$$w^T x + b < 0 => y = -1$$

При этом, оптимальной гиперплоскостью является та, которая максимизирует ширину полосы (зазор) между классами, при этом сама находится по середине этой полосы.

Ставится следующая задача оптимизации:

$$M = \frac{2}{||w||^2} = \frac{2}{w_1^2 + \cdots + w_m^2} \to max$$

$$\frac{||w||^2}{2} \to min$$

При условии:

$$y_i(w^T x + b) \geq 1$$

Функция Лагранжа:

$$L(x, w, b, \alpha) = \frac{1}{2}\sum_{i=1}^{m} w_i^2 - \sum_{i=1}^{n} \alpha_i (y_i(w^T x + b) - 1),$$

$\alpha_i -$ множители Лагранжа.

Необходимые условия седловой точки:

$$-\sum_{i=1}^{n}\alpha_i y_i = 0, \quad w_k - \sum_{i=1}^{n}\alpha_i y_i x_i^k = 0$$

Используем условия седловой точки и переходим к двойственной задаче:

$$\sum_{i=1}^{n}\alpha_i \; -\frac{1}{2}\sum_{i=1}^{n}\sum_{j=1}^{n}\alpha_i\alpha_j y_i y_j x_i^T x_j \;\; \rightarrow max$$

При ограничениях:

$$\alpha_i \geq 0, \qquad \sum_{i=1}^{n}\alpha_i y_i = 0$$

Тогда функция нашей разделяющей гиперплоскости представляется в виде:

$$g(x) = \sum_{i=1}^{n}\alpha_i x_i^T x + b$$

С учётом перехода в пространство большей размерности (для решения задачи при линейно не разделимых данных) функция разделяющей гиперплоскости в новом пространстве имеет вид:

$$g(x) = \sum_{i=1}^{n}\alpha_i \varphi(x_i)^T \varphi(x_i) + b$$

$$g(x) = \sum_{i=1}^{n}\alpha_i K(x_i, x_y) + b, \qquad K\left(x_i, x_j\right) - \text{Ядро}$$

Также требуется ввести значение штрафа, для неверно предсказанных примеров:

$C\sum_{i=1}^{n}\varepsilon_i$, C – штрафной параметр.

**Таким образом, требуется настроить параметр штрафа C, выбрать оптимальное ядро и его параметры.**

### 1.2.2. Наивный байесовский классификатор

В основе метода лежит теорема Байеса, которая связывает апостериорные и априорные вероятности.

Пусть C – множество классов, x – пример, y – метка класса:

$$P(y = c|x) = \frac{P(x|y = c) * P(y = c)}{P(x)}$$

$$c_{opt} = argmax_{c \in \mathrm{C}}\big(P(y = c|x)\big)$$

$$= argmax_{c \in \mathrm{C}}\left(\frac{P(x|y = c) * P(y = c)}{P(x)}\right)$$

Наивность заключается в том, что данный метод предполагает независимость признаков при условии класса.

$$\mathrm{P}(y = c) = \frac{N_c}{N},$$

$$N_c - \text{количество элементов класса C,}$$

$$N - \text{количество элементов в выборке}$$

Класс, к которому относится пример, будем тем, вероятность принадлежности к которому для примера максимальная.

### 1.2.3. Бэггинг

Идея метода заключается в использовании множества слабых классификаторов, которые чуть лучше, чем случайное угадывание, для предсказывания класса для примера, а затем объединение этих предсказаний, путём использования голосования.

Формально:

Обучающая выборка: $D = \{(x_1, y_1), \dots, (x_n, y_n)\}$

Случайно выбираем t элементов из D с возвращением s раз:

$$D_1, \dots, D_s$$

Обучаемся на каждом $D_i$ и получаем последовательность s выходов: $f_1(x), \dots, f_s(x)$.

Получаем итоговый классификатор, который уже будет являться сильным:

$$f(x) = \sum_{i=1}^{s} sign(f_i(x))$$

Математическое ожидание ошибки предсказания:

$$E_{\text{ком}} = E_x\left(\left(\frac{1}{s}\sum_{i=1}^{s}\left(y_i(x) - h(x)\right)\right)^2\right) = E_x\left(\left(\frac{1}{s}\sum_{i=1}^{s}\epsilon_i(x)\right)^2\right)$$

Таким образом, если ошибки не коррелированы, то мы получим ошибку в s раз меньше, тем средняя ошибка всех моделей, но на практике ошибки сильно коррелированы, так как данные в моделях могу совпадать. Однако ошибка сильной модели всегда меньше или равна ошибкам всех слабых моделей.

# 2. Задание 1

## 2.1. SVM

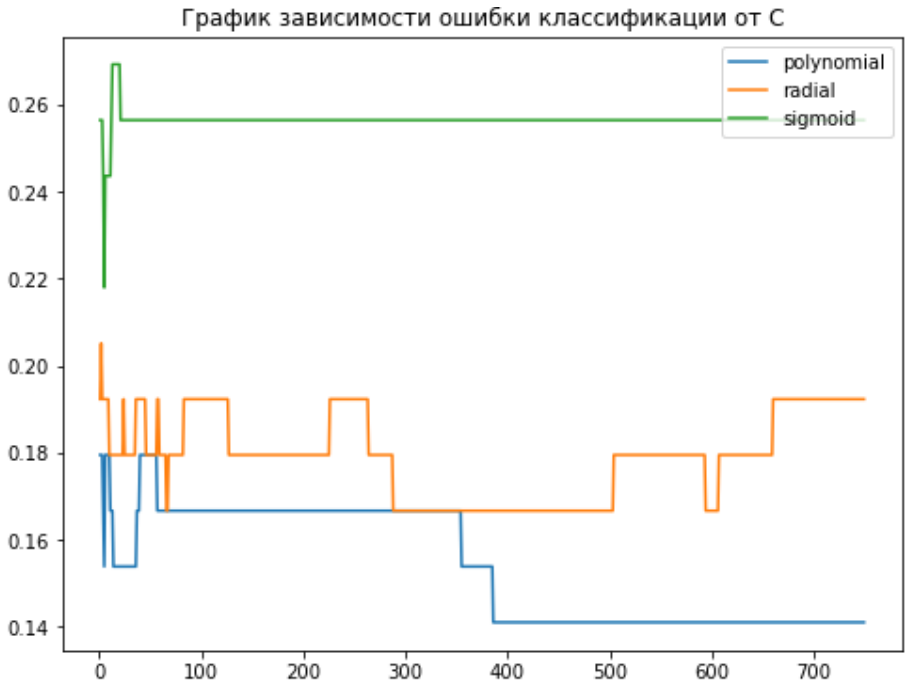
Подбор оптимальных параметров для SVM:

```
kernels = ['poly', 'rbf', 'sigmoid']
for kernel in kernels:
    model = svm.SVC(kernel=kernel, C=1)
    model.fit(X_train, y_train)
    y_predict_test = model.predict(X_test)
    y_predict_train = model.predict(X_train)
    print("Ядро: ", kernel)
    print("Ошибка для тестовой выборки: ", 1-accuracy_score(y_test, y_predict_test))
```

```
Ядро:  poly
Ошибка для тестовой выборки:  0.17948717948717952
Ядро:  rbf
Ошибка для тестовой выборки:  0.1923076923076923
Ядро:  sigmoid
Ошибка для тестовой выборки:  0.2564102564102564
```

```
acc_test_poly = []
acc_test_rbf = []
acc_test_sigmoid = []
c_values = [i for i in range(1, 750)]
for c in c_values:
    for kernel in kernels:
        model = svm.SVC(kernel=kernel, C=c)
        model.fit(X_train, y_train)
        y_predict_test = model.predict(X_test)
        y_predict_train = model.predict(X_train)
        if kernel == 'poly':
            acc_test_poly.append(1-accuracy_score(y_test, y_predict_test))
        if kernel == 'rbf':
            acc_test_rbf.append(1-accuracy_score(y_test, y_predict_test))
        if kernel == 'sigmoid':
            acc_test_sigmoid.append(1-accuracy_score(y_test, y_predict_test))
```

```
fig, ax = plt.subplots(figsize=(8, 6))
ax.plot(c_values, acc_test_poly, label="polynomial")
ax.plot(c_values, acc_test_rbf, label="radial")
ax.plot(c_values, acc_test_sigmoid, label="sigmoid")
ax.set_title("График зависимости ошибки классификации от C")
ax.legend(loc='upper right')
```



Таким образом, оптимальными параметрами для SVM оказались:

Ядро: полиномиальное со степенью 2.

Значение штрафа: 400 (дальше точность выходит на плато).

**Полученная ошибка: 0.141**

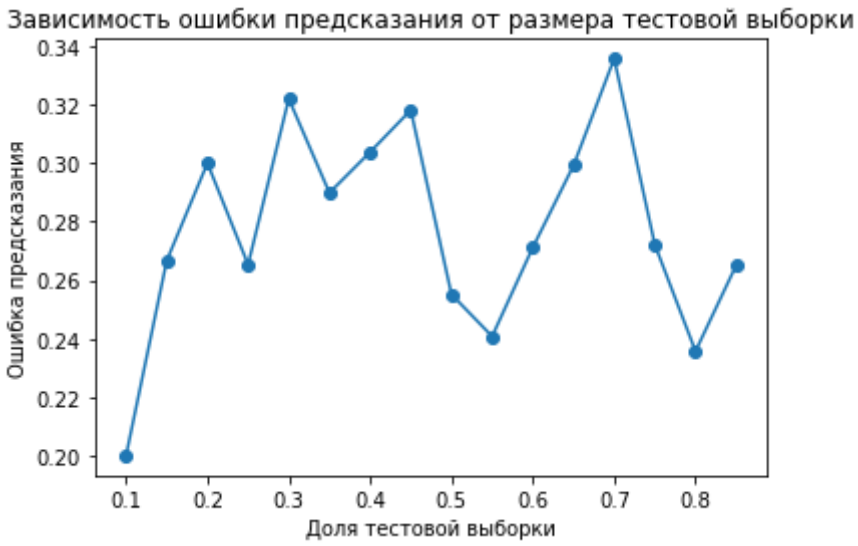
## 2.2. Наивный байесовский классификатор

Среди Bernoulli Naïve Bayes, Multinomial Naive Bayes, Gaussian Naïve Bayes был выбран Gaussian, так как данные представляют собой непрерывные случайные величины.

В качестве настраиваемого параметра: размер выборки.

```
acc = []
test_coef = np.arange(0.1, 0.9, 0.05)
for value in test_coef:
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=value, random_state=35)
    model = GaussianNB()
    model.fit(X_train, y_train)
    y_pred = model.predict(X_test)
    acc.append(1-accuracy_score(y_test, y_pred))
```

```
plt.plot(test_coef, acc, marker='o', linestyle='-')
plt.title('Зависимость ошибки предсказания от размера тестовой выборки')
plt.xlabel('Доля тестовой выборки')
plt.ylabel('Ошибка предсказания')
```




Оптимальная доля тестовой выборки оказалось равной: 0.1

**Полученная минимальная ошибка оказалась равна: 0.2**

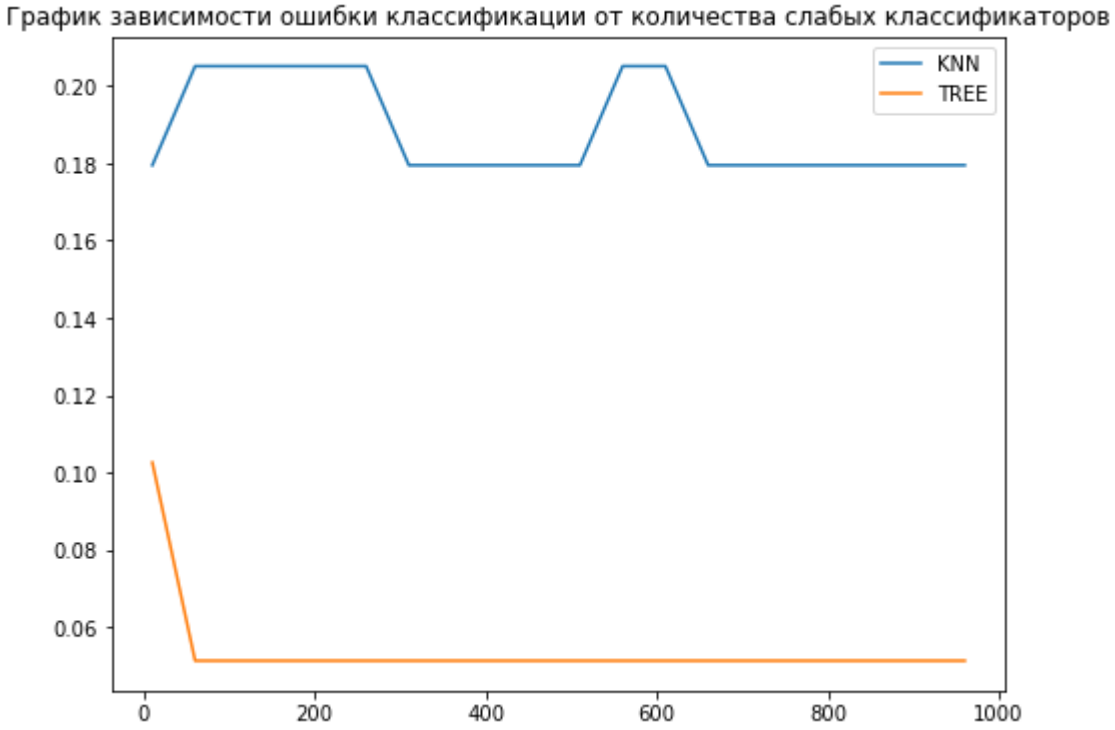
## 2.3. Бэггинг

Для бэггинга можно подбирать слабый классификатор и количество слабых классификаторов.

В качестве модели слабого классификатора будем сравнивать Decision Tree и KNN.

```
acc_test_KNN = []
acc_test_Tree = []
n_estimators = np.arange(10, 1000, 50)
for n in n_estimators:
    bagging_classifier_knn = BaggingClassifier(base_estimator=KNeighborsClassifier(), n_estimators=n, random_state=42)
    bagging_classifier_knn.fit(X_train, y_train)
    y_pred = bagging_classifier_knn.predict(X_test)
    acc_test_KNN.append(1-accuracy_score(y_test, y_pred))
    bagging_classifier_tree = BaggingClassifier(n_estimators=n, random_state=42)
    bagging_classifier_tree.fit(X_train, y_train)
    y_pred = bagging_classifier_tree.predict(X_test)
    acc_test_Tree.append(1-accuracy_score(y_test, y_pred))
```

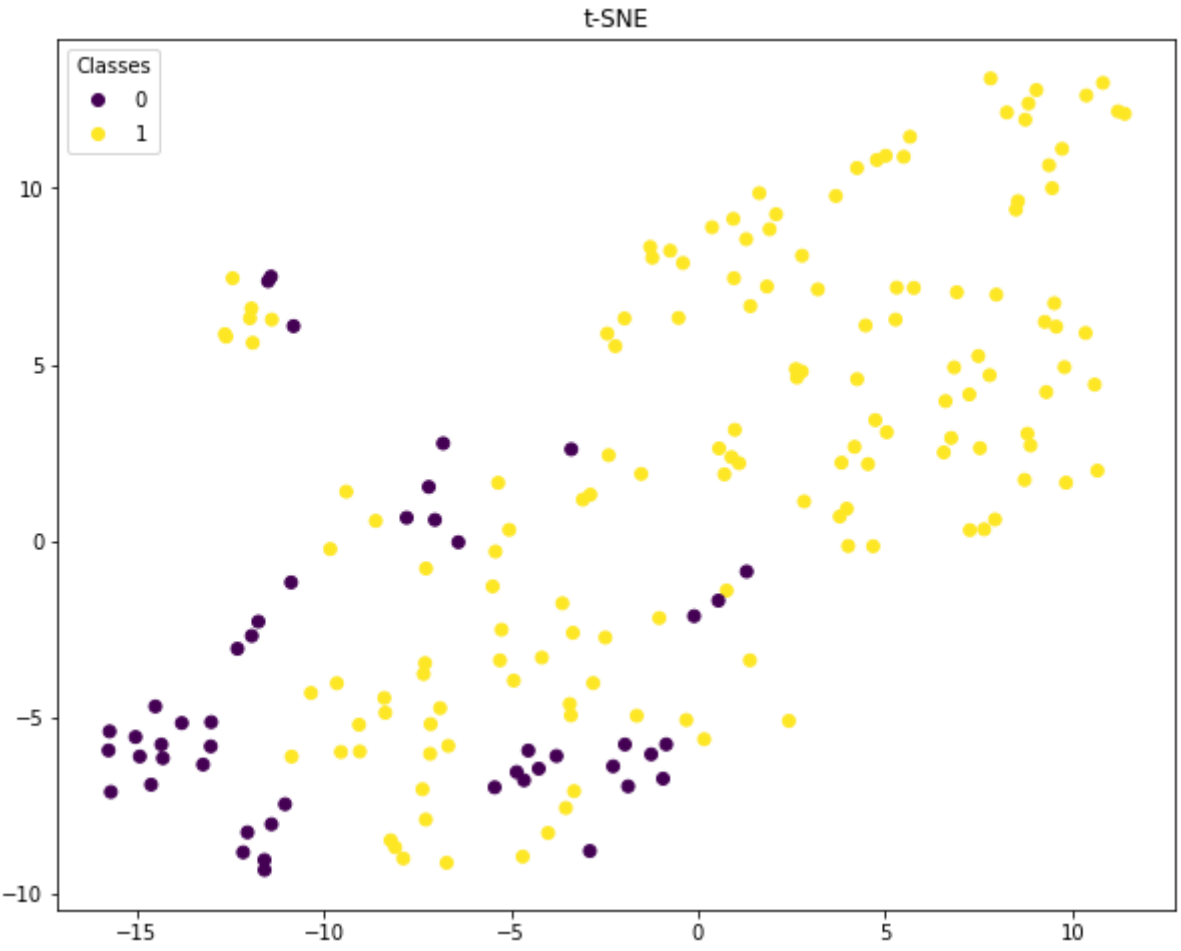
```
fig, ax = plt.subplots(figsize=(8, 6))
ax.plot(n_estimators, acc_test_KNN, label="KNN")
ax.plot(n_estimators, acc_test_Tree, label="TREE")
ax.set_title("График зависимости ошибки классификации от количества слабых классификаторов")
ax.legend(loc='upper right')
```




Таким образом, лучше оказалась модель со слабым классификатором - Decision Tree, которая после количества слабых классификаторов равного 40 выходит на плато, достигая **минимальной ошибки классификации = 0.051.**

## 2.4. t-SNE

```
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
tsne = TSNE(n_components=2, random_state=42)
X_tsne = tsne.fit_transform(X_train_scaled)
tsne_df = pd.DataFrame(data=X_tsne, columns=['Dimension 1', 'Dimension 2'])
tsne_df['Target'] = y_train.values
plt.figure(figsize=(10,8))
scatter=plt.scatter(tsne_df['Dimension 1'], tsne_df['Dimension 2'], c=tsne_df['Target'], cmap='viridis')
plt.legend(*scatter.legend_elements(), title='Classes')
plt.title('t-SNE')
plt.show()
```


t-SNE
Classes
0
1
10
5
0
−5
−10
−15
−10
−5
0
5
10

## 3. Задание 2

Исходя из результатов, полученных в предыдущем задание, наилучшая модель по вероятности ошибочной классификации на тестовых данных, ожидаемо, - Бэггинг, построенный на 40 деревьях решений.

# 4. Задание 3

## 4.1. Описание метода k-средних

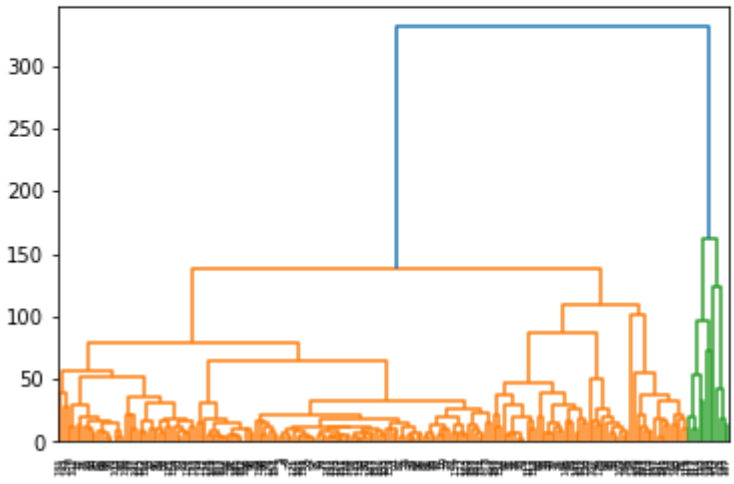
1. Случайно разбиваем объекты на $K$ кластеров.
2. Вычисляем центры тяжести $m_k$ кластеров:

$$m_k = \frac{1}{N_k} \sum_{x_i \in C_k} x_i \ , k = 1, \dots, K$$

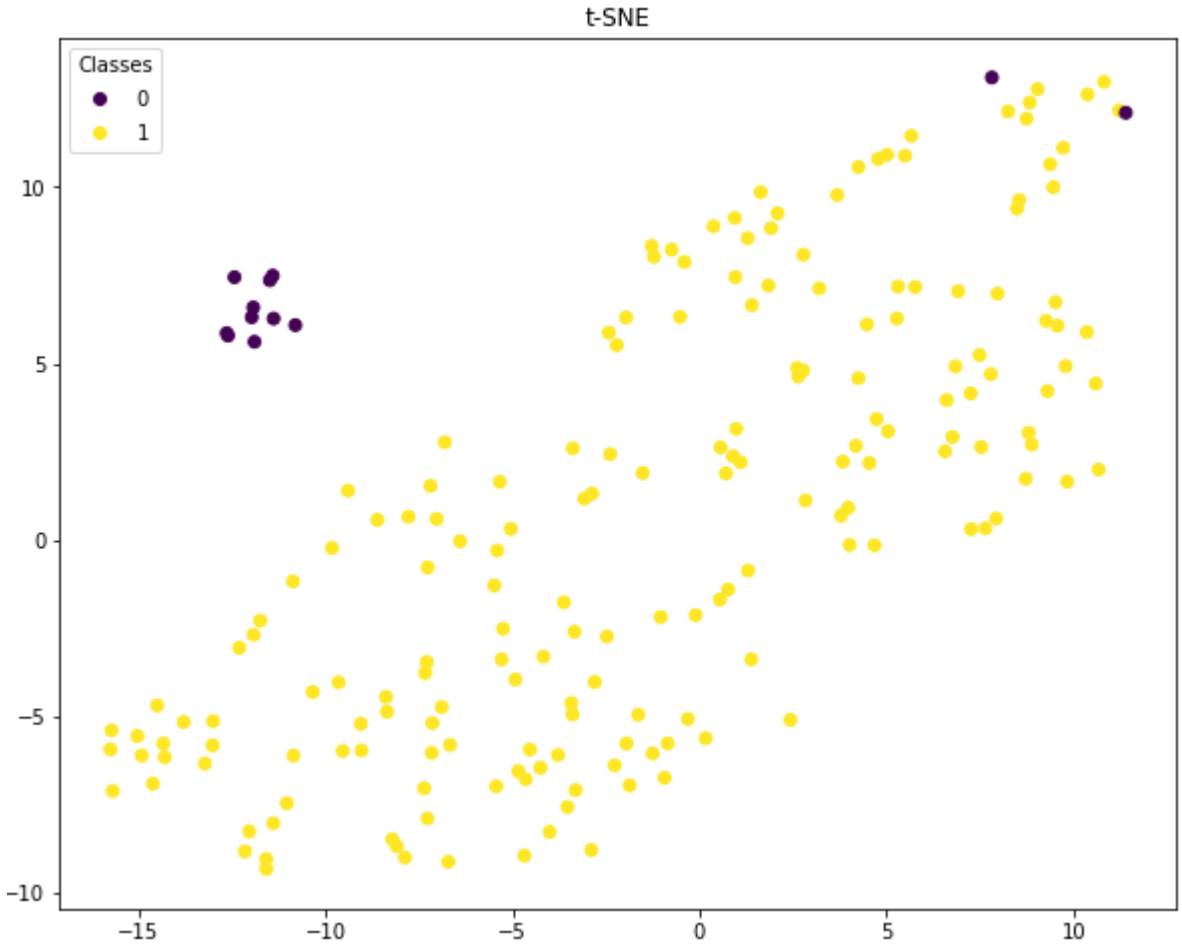
3. Вычисляем расстояния $\rho(x_i, m_k)$ от точки $x_i$ до всех $m_k$. Записываем в тот класс, расстояние до центра тяжести которого минимальное.
4. Повторяем шаг 3 для всех $x_i$
5. Если хотя бы один кластер изменился, то переходим на шаг 2, иначе завершение.

## 4.2 Реализация

```
data = data.drop('status',axis=1)
Z = hierarchy.linkage(data, 'average')
plt.figure()
dn = hierarchy.dendrogram(Z)
```




```
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X)
tsne = TSNE(n_components=2, random_state=42)
X_tsne = tsne.fit_transform(X_train_scaled)
tsne_df = pd.DataFrame(data=X_tsne, columns=['Dimension 1', 'Dimension 2'])
tsne_df['Target'] = kmeans.labels_
plt.figure(figsize=(10,8))
scatter=plt.scatter(tsne_df['Dimension 1'], tsne_df['Dimension 2'], c=tsne_df['Target'], cmap='viridis')
plt.legend(*scatter.legend_elements(), title='Classes')
plt.title('t-SNE')
plt.show()
```




**Ошибка кластеризации: 0.277**

# 5. Задание 4

## 5.1 Описание метода Лассо

Математическое ожидание отклонения имеет вид:

$$E(b) = \sum_{i=1}^{n} \left(y_i - f_i(x_i, b)\right)^2,$$

где $b$ – коэффициенты линейной функции.

Логистическая регрессия получается в результате использования логарифмического функционала риска:

$$E(b) = \sum_{i=1}^{n} \log_2\left(1 + e^{(y_i f(x_i, b))}\right)$$

Это эквивалентно замене $f_i(x_i, b)$ логистической или сигмоидной функцией:

$$g(z) = \frac{1}{1+e^{-z}}, \quad z = b^T x, \qquad 0 \le g(z) \le 1$$

Получим вероятности для классификации:

$$P(y = 0|x, b) = \frac{1}{1+e^{-b^T x}}$$

$$P(y = 1|x, b) = \frac{e^{-b^T x}}{1+e^{-b^T x}}$$

Ищем коэффициенты $b_i$, как ОМП оценки:

$$L(y|x, b) = \prod_{i=1}^{n} \left(1 - g(b^T x_i)\right)^{y_i} g(b^T x_i)^{1-y_i} \rightarrow \max_b$$

Берём отрицательный логарифм функции правдоподобия, накладываем ограничения на коэффициенты $b_i$. Получаем требуемую модель:

Ограничения на $b_i$:

$$\sum_{j=1}^{m} |b_j| < C$$

Модель:

$$\min_b [-\frac{1}{n} \sum_{i=1}^{n} y_i b^T x_i - \ln\left(1 + e^{-b^T x_i}\right)] + \alpha \sum_{j=1}^{m} |b_j|$$

## 5.2 Реализация

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

lasso_model = LogisticRegression(penalty='l1', solver='liblinear', C=1.0)
lasso_model.fit(X, y)

coefficients = lasso_model.coef_[0]
feature_names = X.columns
coefficients_df = pd.DataFrame({'Feature': feature_names, 'Coefficient': coefficients})
coefficients_df['Abs_Coefficient'] = np.abs(coefficients_df['Coefficient'])
coefficients_df = coefficients_df.sort_values(by='Abs_Coefficient', ascending=False)

print("Top Significant Features:")
print(coefficients_df[['Feature', 'Coefficient']])
```

```
Top Significant Features:
             Feature  Coefficient
20                D2     3.381114
9   MDVP:Shimmer(dB)     2.583487
18           spread1     1.397157
15               HNR     0.162981
0         MDVP:Fo(Hz)    -0.009267
1        MDVP:Fhi(Hz)    -0.003444
2        MDVP:Flo(Hz)    -0.002575
13        Shimmer:DDA     0.000000
19           spread2     0.000000
17               DFA     0.000000
16              RPDE     0.000000
14               NHR     0.000000
11       Shimmer:APQ5     0.000000
12           MDVP:APQ     0.000000
10       Shimmer:APQ3     0.000000
8        MDVP:Shimmer     0.000000
7          Jitter:DDP     0.000000
6            MDVP:PPQ     0.000000
5            MDVP:RAP     0.000000
4   MDVP:Jitter(Abs)     0.000000
3     MDVP:Jitter(%)     0.000000
21               PPE     0.000000
```

Параметры, у которых значения коэффициента равны 0, являются не влияющими на результат. Чем выше коэффициент, тем выше влияние.

## 6. Задание 5

Используем автокодер для понижения размерности до 7. Достаём закодированные данные из скрытого слоя и применяем бэггинг из первого задания.

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
input_layer = Input(shape=(22,))
encoded = Dense(7, activation='relu')(input_layer)
decoded = Dense(22,activation='linear')(encoded)
autoencoder = Model(input_layer, decoded)
autoencoder.compile(optimizer='adam', loss='mean_squared_error')
autoencoder.fit(X_train, X_train, epochs=70, batch_size=8, shuffle=True, validation_data=(X_test, X_test))
encoder = Model(input_layer, encoded)
encoded_data = encoder.predict(X)
```

```
X_train, X_test, y_train, y_test = train_test_split(encoded_data, y, test_size=0.4, random_state=42)
bagging_classifier_tree = BaggingClassifier(n_estimators=40, random_state=42)
bagging_classifier_tree.fit(X_train, y_train)
y_pred = bagging_classifier_tree.predict(X_test)
print(1-accuracy_score(y_test, y_pred))
```

Получаем, что **точность ухудшилась и стала равной 0.19**.

Теперь используем регуляризацию L1, чтобы занулить выходы некоторых нейронов внутреннего слоя, чтобы сократить размерность.

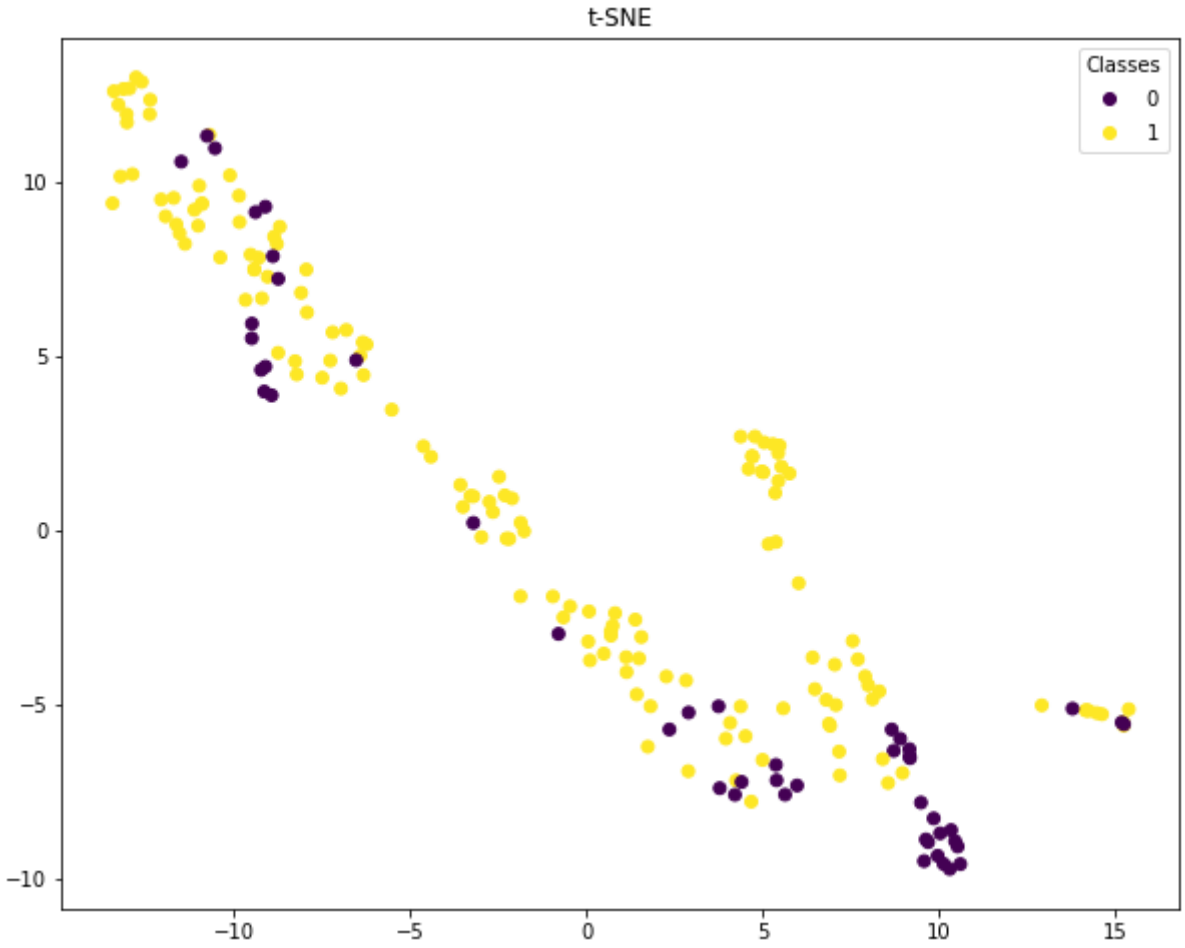
```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
input_layer = Input(shape=(22,))
encoded = Dense(22, activation='relu', activity_regularizer=tf.keras.regularizers.l1(1e-4))(input_layer)
decoded = Dense(22,activation='linear')(encoded)
autoencoder = Model(input_layer, decoded)
autoencoder.compile(optimizer='adam', loss='mean_squared_error')
autoencoder.fit(X_train, X_train, epochs=70, batch_size=8, shuffle=True, validation_data=(X_test, X_test))
encoder = Model(input_layer, encoded)
encoded_data = encoder.predict(X)
```

Используем также бэггинг из первого задания.

```
X_train, X_test, y_train, y_test = train_test_split(encoded_data, y, test_size=0.4, random_state=42)
bagging_classifier_tree = BaggingClassifier(n_estimators=40, random_state=42)
bagging_classifier_tree.fit(X_train, y_train)
y_pred = bagging_classifier_tree.predict(X_test)
print(1-accuracy_score(y_test, y_pred))
```

**Точность ухудшилась и стала равной 0.23.**

В итоге, предпочтительнее оказалось уменьшение скрытых слоёв, чем разряженность скрытого слоя.




Зашумленный автокодер используется для того, чтобы помогать избавляться от шумов во входных данных. Для этого во входные данные вносится гауссовский шум.

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
X_train_noisy = X_train + 0.2*np.random.normal(loc=0.0, scale=1.0, size=X_train.shape)
X_test_noisy =  X_test + 0.2*np.random.normal(loc=0.0, scale=1.0, size=X_test.shape)
input_layer = Input(shape=(22,))
encoded = Dense(7, activation='relu')(input_layer)
decoded = Dense(22,activation='linear')(encoded)
autoencoder = Model(input_layer, decoded)
autoencoder.compile(optimizer='adam', loss='mean_squared_error')
autoencoder.fit(X_train_noisy, X_train, epochs=70, batch_size=8, shuffle=True, validation_data=(X_test_noisy, X_test))
encoder = Model(input_layer, encoded)
```

Для зашумленного автокодера точность оказалось равной 0.25, что является худшим результатом среди рассмотренных автокодеров.