МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра ВТ

ОТЧЕТ

по лабораторной работе № 1

по дисциплине «Машинное обучение»

Классификация

Студент гр. 9308 _____ Хамитов А.К.

_____

Преподаватель _____ Беляев П. Ю.

Санкт-Петербург

2022

Цель работы

Получения и закрепления навыков предобработки данных и применения методов машинного обучения для решения задач классификации.

Выбрать набор данных на сайте Kaggle.com, такой, что:

1. Число столбцов признаков – не менее 10;
2. Число записей – не менее 10000.

Был выбран датасет:
https://www.kaggle.com/datasets/aswathrao/restaurant-quality-analysis

Ход работы

Анализ данных и проектирование признаков

Датасет имеет следующие столбцы, а состоит из 1100000 тысяч строк

```
In [7]: df.dtypes

Out[7]: ID                      int64
        Date                   object
        LicenseNo               int64
        Assessment_ID           int64
        Assessment_Name         int64
        Restaurent Type        object
        Street ID               int64
        City ID                object
        State ID               object
        LocationID            float64
        Reason                 object
        SectionViolations     float64
        Risk_level             object
        Geographical_Location  object
        Assessment_Results      int64
        dtype: object
```

Первым делом были убраны дубликаты и нерелевантные значения.

```
In [14]: df = df.drop_duplicates()
         df.head(5)
```

Также, чтобы модели обучения смогли воспринимать данные в виде строк, последние были переконвертированы в числа. Представлено на рисунке ниже.

Замена категориальных переменных

```
In [19]: df['Risk_level'].value_counts()

Out[19]: High      36358
         Medium     9413
         Low        3083
         Name: Risk_level, dtype: int64

In [20]: num_replace = {'Risk_level': {'High': 3, 'Medium': 2, 'Low': 1}, 'Reason': {}, 'RestaurentType': {}}

         freq = df['Reason'].value_counts()
         for index in freq.index:
             num_replace['Reason'][index] = freq[index]

         freq = df['RestaurentType'].value_counts()
         for index in freq.index:
             num_replace['RestaurentType'][index] = freq[index]

         df = df.replace(num_replace)

         df.head()

Out[20]:
```

В датасете удалены выбросы

```
In [24]: columns_int = df.select_dtypes(include=["int64", "float64"])
```

```
In [25]: for x in columns_int:
             q75,q25 = np.percentile(df.loc[:,x],[75,25])
             intr_qr = q75-q25

             max = q75+(1.5*intr_qr)
             min = q25-(1.5*intr_qr)

             df.loc[df[x] < min,x] = np.nan
             df.loc[df[x] > max,x] = np.nan
```

```
In [26]: df.isnull().sum()
```

```
Out[26]: Assessment_Name            0
         RestaurentType             0
         Street ID                  0
         LocationID               347
         Reason                     0
         SectionViolations          2
         Risk_level                 0
         Geographical_Location      0
         Assessment_Results         0
         dtype: int64
```

```
In [27]: df = df.dropna()
         df.isnull().sum()
```

```
Out[27]: Assessment_Name          0
         RestaurentType           0
         Street ID                0
         LocationID               0
         Reason                   0
         SectionViolations        0
         Risk_level               0
         Geographical_Location    0
         Assessment_Results       0
         dtype: int64
```

И проведена нормализация:

Нормализация данных

```
In [28]: # df.loc[:, df.columns != 'Assessment_Results']
         df['Geographical_Location'] = df['Geographical_Location'].astype(float)
         df.dtypes
```

```
Out[28]: Assessment_Name          float64
         RestaurentType           float64
         Street ID                float64
         LocationID               float64
         Reason                   float64
         SectionViolations        float64
         Risk_level               float64
         Geographical_Location    float64
         Assessment_Results       float64
         dtype: object
```

```
In [29]: cdf = df
         cdf = (df-df.min ())/(df.max ()-df.min ())
         scaled_df = cdf
```

Обучение и прогноз

После обработки и исследования признаков датасета, были обучены модели: логистическая регрессия, k-ближайших соседей, машина

опорных векторов, наивный байесовский классификатор, дерево решений, случайный лес, XGBoost.

Получились следующие результаты:

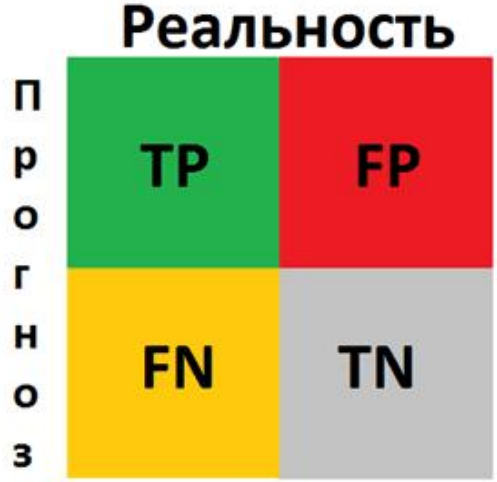
Модель	precision	recall	f1-score
Логистическая регрессия	0.68	0.68	0.68
K-ближайших соседей	0.75	0.75	0.75
Машина опорных векторов	0.75	0.75	0.75
Наивный байесовский классификатор	0.74	0.74	0.74
Дерево решений	0.81	0.81	0.81
Случайный лес	0.87	0.87	0.87
XGBoost	0.88	0.88	0.88

Для моделей были построены ROC-кривые. Также для некоторых моделей были подобраны параметры из представленных с помощью Search CV.

ROC-кривые – это зависимость TPR от FPR. С помощью этой кривой можно примерно прикинуть насколько хорошо справляется обученная модель.

Confusion matrix - это матрица, показывающая количество попаданий в TP, FP, FN и TN.

Прогноз	Реальность	
	+	–
+	*True Positive (истинно-положительное решение)*: прогноз совпал с реальностью, результат положительный произошел, как и было предсказано ML-моделью	*False Positive (ложноположительное решение)*: ошибка 1-го рода, ML-модель предсказала положительный результат, а на самом деле он отрицательный
–	*False Negative (ложноотрицательное решение)*: ошибка 2-го рода – ML-модель предсказала отрицательный результат, но на самом деле он положительный	*True Negative (истинно-отрицательное решение)*: результат отрицательный, ML-прогноз совпал с реальностью



Precision показывает сколько в процентах было предсказано результатов верно.

TP/(TP+FP)

Recall показывает сколько было получено в процентах TP.

TP/(TP+FN)

f1-score показывает проценты верных позитивных предсказаний.

2*(Recall * Precision)/(Recall + Precision)

Во время разбивки и подготовки датасета для обучения не был задействован SMOTE fit_resample, так как предсказывающий признак и так распределён поровну.

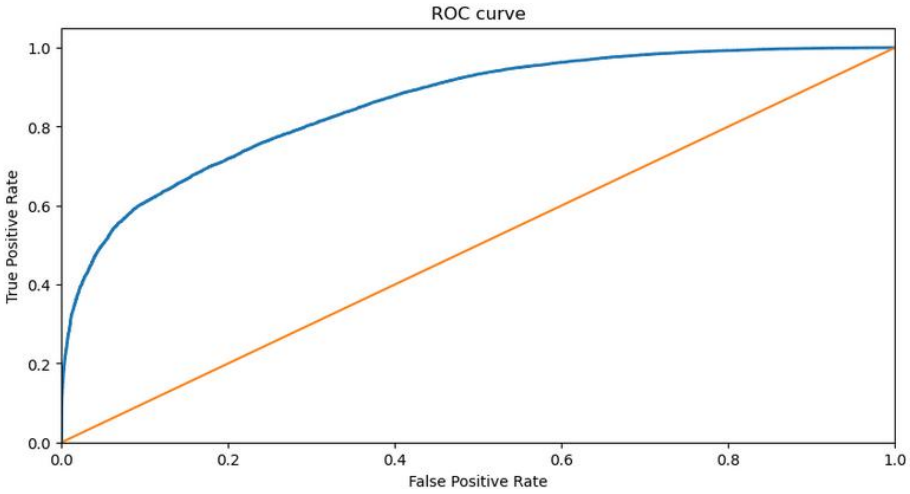
Анализируя полученные результаты, можно сделать вывод, что с данным датасетом лучше всего справилась модель XGBoost, а хуже всех – наивный байесовский классификатор (много зависимых признаков в исследуемой области).

```
def do_fit_and_predict(model, dataframe) -> "(predicts, class_report, conf_matrix)":
    df = dataframe.copy()

    X = df.copy()
    Y = X.pop('round_winner')

    #X, y = SMOTE().fit_resample(X, y) # Не нужен. И так поровну в "round_winner"

    X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size=0.33, random_state=5051)

    sc = StandardScaler()
    X_train = sc.fit_transform(X_train)
    X_test = sc.transform(X_test)

    #model = LogisticRegression()
    model.fit(X_train, Y_train)

    Y_pred = model.predict(X_test)

    class_report = classification_report(Y_test, Y_pred)
    conf_matrix = confusion_matrix(Y_test, Y_pred)

    return (Y_pred, class_report, conf_matrix)
```

Логистическая регрессия – это самый простой в понимании метод машинного обучения. Модель представляет из себя «линейную интерполяцию». Просто подбираются коэффициенты в уравнение вида:

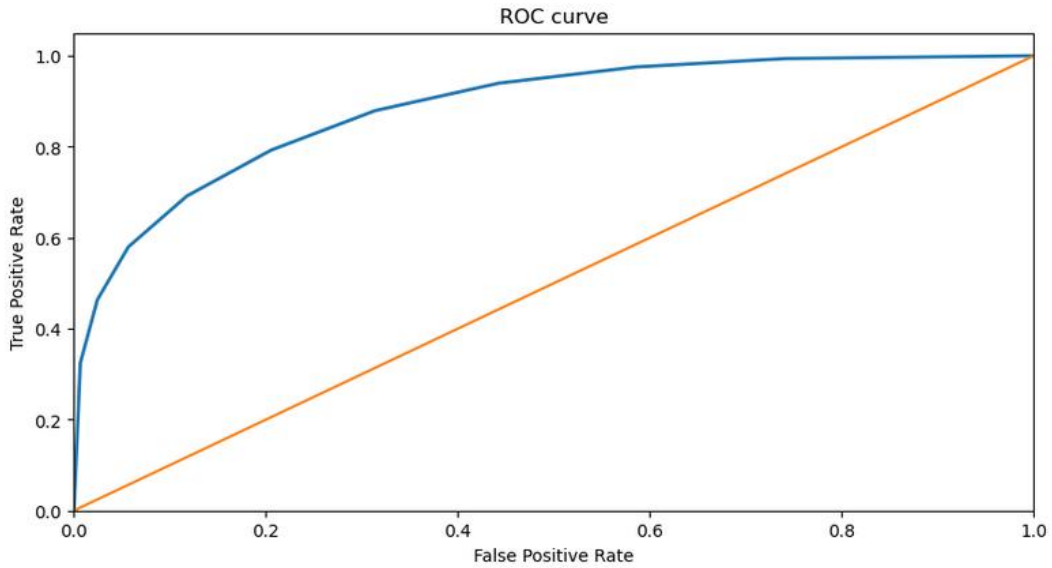
$$f(\mathbf{X}) = \beta_0 + X_1 \times \beta_1 + X_2 \times \beta_2 + \ldots$$

```
model = LogisticRegression()
preds, class_report, conf_matrix = do_fit_and_predict(model, df_ints)

print(conf_matrix)
print(class_report)
```



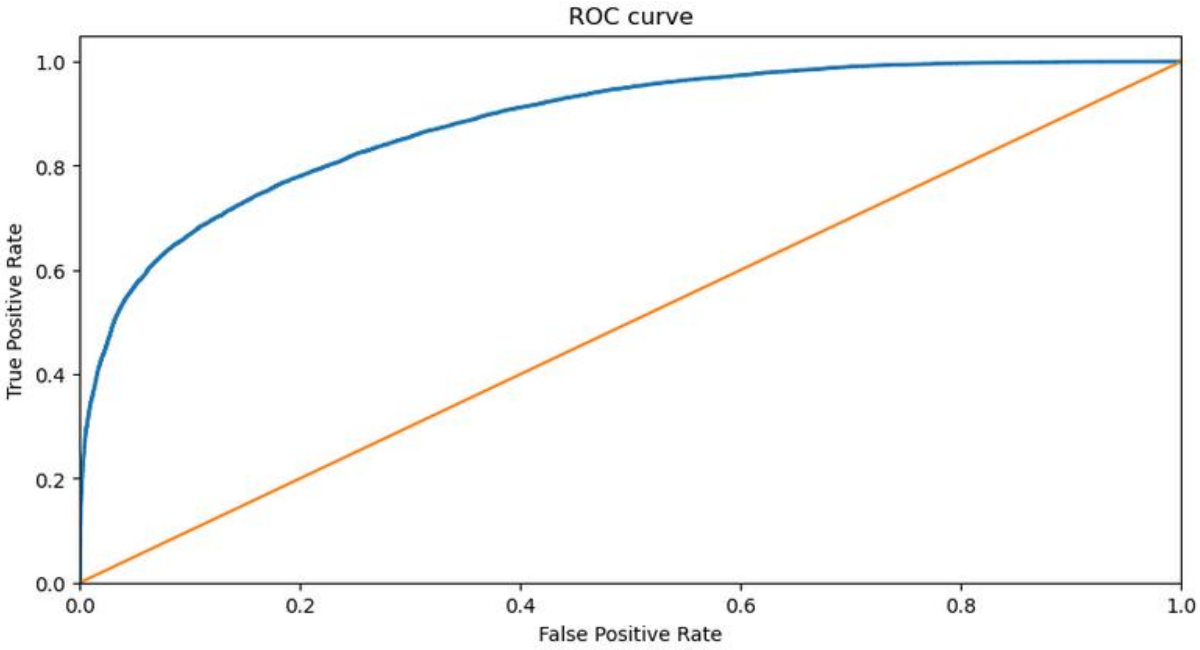
K-ближайших соседей – тоже простой метод, но в отличие от логистической регрессии, он уже «нелинейный». Суть состоит в том, что вычисляются длина вектора в пространстве признаков. Далее сравнивается с остальными, выбираются ближайшие соседи, и в зависимости от количества соседей определённого класса выносится предсказание. Расстояние в пространстве признаков можно задать по-разному, например, евклидовое или манхэттенское.

```
# {'n_neighbors': 9}

if(knn_bp == None):
    model = KNeighborsClassifier()
else:
    model = KNeighborsClassifier(**knn_bp)
preds, class_report, conf_matrix = do_fit_and_predict(model, df_ints)

print(conf_matrix)
print(class_report)
```



Машина опорных векторов хоть и более «продвинутая» модель, но с данным датасетом сравилась не сильно лучше k-ближайших соседей. Получается вычислительные мощности можно было бы и не тратить, а использовать, k-ближайших соседей, если такая точность модели устраивает. Суть метода заключается в поиске гиперплоскости, оптимально разделяющих 2 класса. Метод тоже нелинейный, но это только лишь из-за того, что выбрано «нелинейное» ядро. Ядро «включает» в себя пространство признаков, там и строится «линейная» граница, которая после возвращение в исходное пространство оказывается нелинейной.

```
model = SVC()
preds, class_report, conf_matrix = do_fit_and_predict(model, df_ints)

print(conf_matrix)
print(class_report)
```



Наивный байесовский классификатор – один из простейших метод. Он «предполагает», что признаки независимы между собой. Как видно в данном датасете такое «предполагать» было неправильно. Потом с помощью теоремы Байеса считаются вероятности, что данная «строка» является n-ым классом, затем выбирается наибольшая. Чтобы при подсчётах не получались огромные числа, используют логарифмы, и тогда выбирается наименьшее число.

```
from collections import defaultdict

class NaiveBayesClassifier():

# C - множество классов
# X - множество признаков
# x = argmax(f(x)) - x, при котором f(x) максимальный
#
# Теорема Байеса
# P(c|x0,...,xn) = ( P(x0,...,xn|c) * P(c) ) / P(x0,...,xn)
#
# Нужно найти argmax(P(c|x0,...,xn)),
#                       где c принадлежит C, xi принадлежит X
# argmax(  P(c|x0,...,xn)  ) = [ по т. Байеса] =
# = argmax(  P(x0,...,xn|c) * P(c) / P(x0,...,xn)  ) =
# = [надо найти максимум, а так как P(x0,...,xn) = const, значит
#      отбрасываем эту константу, на поиск максимума она не повлияет] =
# = argmax(  P(x0,...,xn|c) * P(c)  )
#
# "наивный" в названии означает, что x0,...,xn независимы, то есть:
#      P(x0,...,xn|c) = P(x0|c)*...*P(xn|c)
#                  но тогда будут очень большие числа. Поэтому log:
#  P(x0|c)*...*P(xn|c)*P(c) ~ -log(P(x0|c))+...+-log(P(xn|c))+-log(P(c))
#    Если log, то не argmax, а argmin
#    Если log, то не "*", а "+"
#
# Нахождение P(c) и всех P(xi|c) (точнее их логарифмов) и есть обучение
# Предсказание - это
#              argmin( -ln(P(c)) + -ln(P(x0|c)) + ... + -ln(P(xn|c)))


    def __init__(self):
        # Вероятность встретить класс ~ P(c)
        self.__class_freq = defaultdict(lambda:0)

        # Вероятность P(xi|c)
        self.__feat_freq = defaultdict(lambda:0)


    def fit(self, X, y):
        # calculate classes and features frequencies
# zip(X, y) is [((x00, ..., xn0), y0), ..., ((x0n, ..., xnn), yn)]
        for feature, label in zip(X, y):
            self.__class_freq[label] += 1
            for value in feature:
                self.__feat_freq[(value, label)] += 1

        # normalizate values
        num_samples = len(X)
        for k in self.__class_freq:
            self.__class_freq[k] /= num_samples

        for value, label in self.__feat_freq:
            self.__feat_freq[(value, label)] /= self.__class_freq[label]

        return self

    def _predict(self, X):
        # return argmin of classes
        return min(self.__class_freq.keys(),
                   key=lambda c : self.__calculate_class_freq(X, c))

    def predict(self, X):
        return [self._predict(x) for x in X]

    def __calculate_class_freq(self, X, clss):
        # calculate frequence for current class
        freq = - np.log(self.__class_freq[clss])

        for feat in X:
            freq += - np.log(self.__feat_freq.get((feat, clss), 10 ** (-7)))
        return freq
```

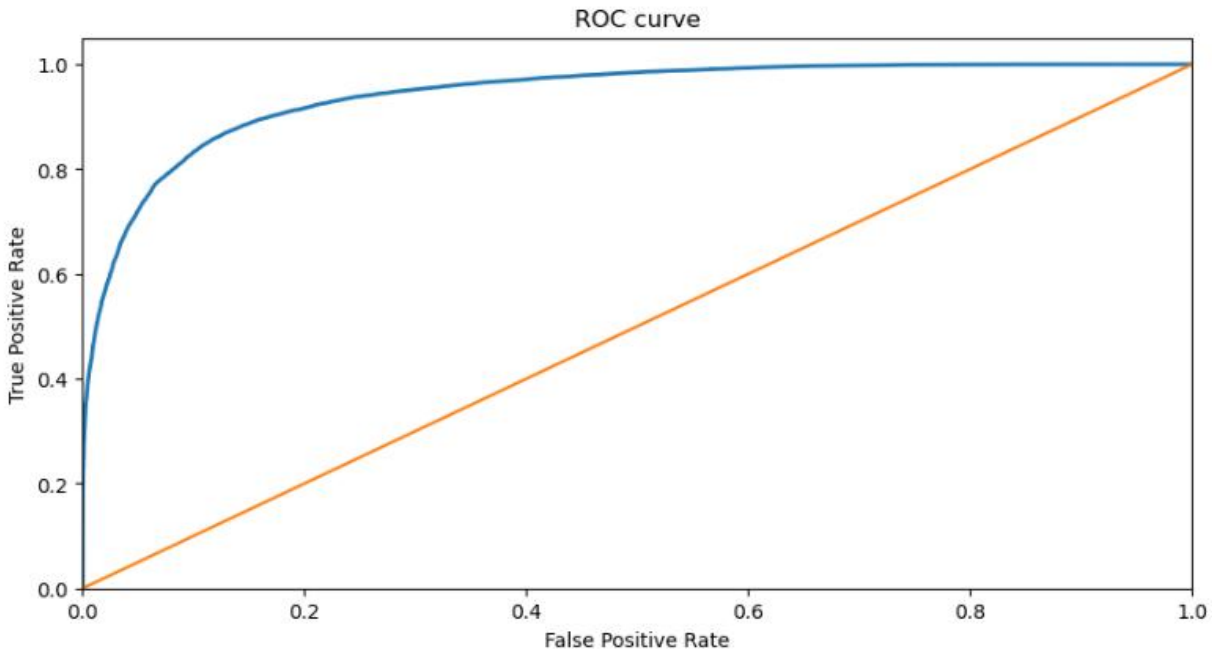
```
preds, class_report, conf_matrix = do_fit_and_predict(NaiveBayesClassifier(), df_ints)

print(conf_matrix)
print(class_report)
```

Дерево решений пытается делить данные таким образом, чтобы после разделение уменьшилась энтропия. Здесь главное не переборщить, чтобы не произошло переобучение. Вместо энтропии можно использовать и другие критерии качества разбиения, например, неопределённость Джини или ошибку классификации.

```
model = DecisionTreeClassifier()
preds, class_report, conf_matrix = do_fit_and_predict(DecisionTreeClassifier(), df_ints)

print(conf_matrix)
print(class_report)
```

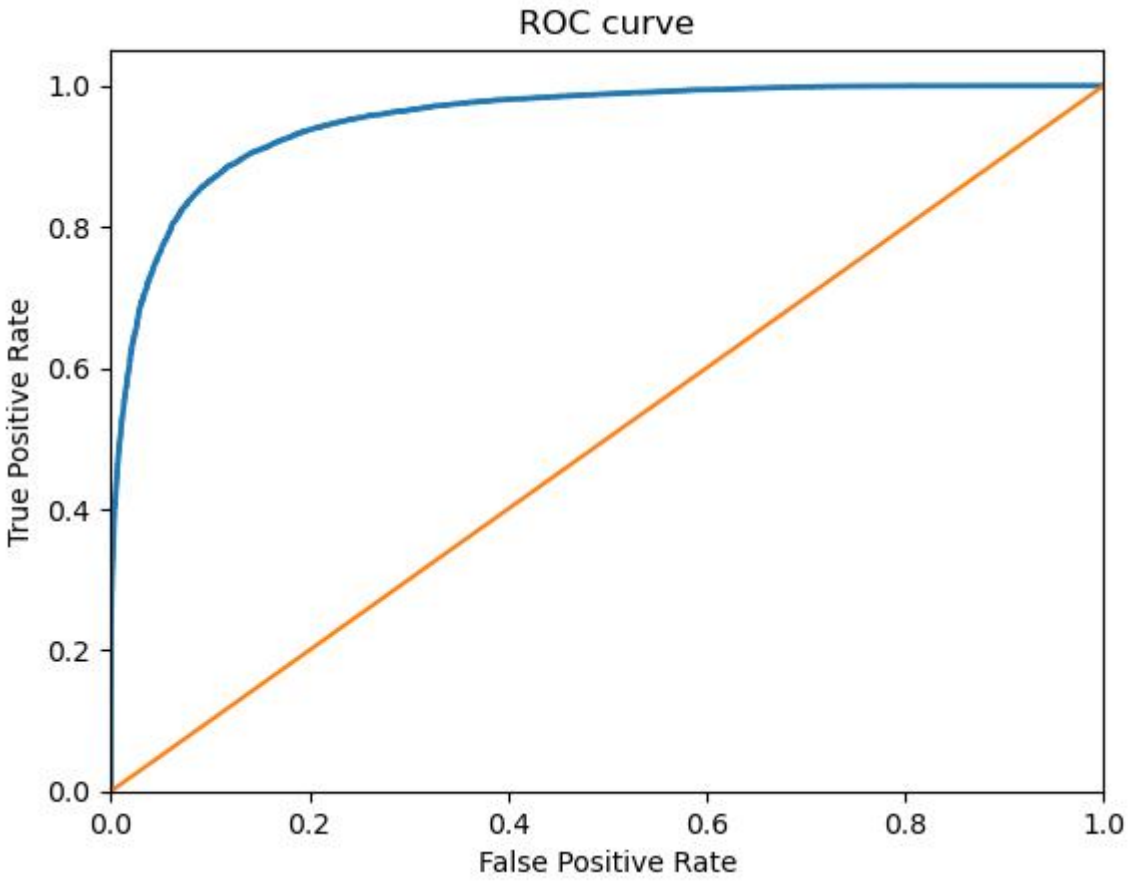
Случайный лес – это тот случай, когда «одно дерево хорошо, а N – лучше». Теперь строится целый лес деревьев решений, они все выносят предсказание, и выбирается голосованием итоговое.

```
## лес
# {'criterion': 'entropy', 'max_depth': 100, 'n_estimators': 1000, 'random_state': 5051}
model = RandomForestClassifier(**rf_bp)
model.fit(X_train, Y_train)

Y_pred = model.predict(X_test)

class_report = classification_report(Y_test, Y_pred)
conf_matrix = confusion_matrix(Y_test, Y_pred)

print(conf_matrix)
print(class_report)
```



XGBoost – это метод, который позволяет строить, например, лес по-другому. XGBoost градиентно бустит деревья решений. XGBoost строит ансамбль слабо-предсказывающих моделей (например, деревьев решений). Здесь обучение модели последовательное, то есть новое звено ансамбля добавляется с учётом отклонения ансамбля до добавления. И так пока ошибка уменьшается или не выполнится условие остановки.

```
# {'subsample': 1.0, 'random_state': 228, 'nthread': 1, 'n_estimators': 500, 'min_child_we

model = XGBClassifier(objective='binary:logistic', **xgb_bp)
model.fit(X_train, Y_train)
Y_pred = model.predict(X_test)

class_report = classification_report(Y_test, Y_pred)
conf_matrix = confusion_matrix(Y_test, Y_pred)

print(conf_matrix)
print(class_report)
```



Для данного датасета метод XGBoost показал наилучшие результаты.

Выводы

В ходе выполнения лабораторной работы были получены и закреплены навыки предобработки данных и применения методов машинного обучения для решения задач классификации на конкретном датасете.