

Машинное обучение с использованием библиотек Python

С.В. Лемешевский (sergey.lemeshevsky@gmail.com)

Институт математики НАН Беларуси

May 4, 2020

Содержание

| | |
|---|-----------|
| 1 Основные определения и постановки задач машинного обучения | 1 |
| 1.1 Примеры задач машинного обучения | 1 |
| 1.2 Линейная регрессия | 2 |
| 1.3 Загрузка данных | 3 |
| 1.4 Первая модель | 5 |
| 1.5 Работа с категориальными признаками | 7 |
| 2 Предобработка данных | 10 |
| 2.1 Работа с текстовыми данными | 10 |
| 2.2 Трансформация признаков и целевой переменной | 13 |
| 2.3 Бинаризация | 14 |
| 2.4 Транзакционные данные | 15 |
| 3 Простые модели классификации | 16 |
| 3.1 Матрица ошибок | 16 |
| 3.2 Линейная классификация | 17 |
| 4 Задание | 20 |
| 4.1 Задание по базе wine | 20 |

1. Основные определения и постановки задач машинного обучения

Машинное обучение — это раздел математики, изучающий способы извлечения закономерностей из ограниченного числа примеров.

1.1. Примеры задач машинного обучения

Рассмотрим несколько примеров задач, которые решаются с помощью машинного обучения.

Кредитный скоринг. *Задача:* выяснить, какие заявки на кредит можно одобрить.

Лента Facebook/Дзен по интересности (вместо сортировки по времени). *Задача:* показать посты, наиболее интересные для конкретного человека.

Детектирование некорректной работы. Предположим, что у нас есть завод, на котором происходят некоторые процессы (стоят какие-то котлы, станки, работают сотрудники). На предприятии может произойти поломка, например, сломается датчик уровня жидкости в баке, из-за чего насос не остановится при достижении нужного уровня и нефть начнёт разливаться по полу, что может привести к неизвестным последствиям. Или же сотрудники объявят забастовку и вся работа остановится. Мы хотим, чтобы завод работал исправно, а обо всех проблемах узнавать как можно раньше.

Задача: предсказать поломки/нештатные ситуации на заводе.

Вопросно-ответная система (как Siri). *Задача:* ответить голосом на вопрос, заданный голосом.

Self-driving cars. *Задача:* доехать из точки в точку.

Перенос стиля изображения. *Задача:* перенести стиль одного изображения на другое (смешать стиль одного с контекстом другого).

Как видим, задачи очень разнообразны. Мы начнем наш путь со следующей классической постановки (к которой, кстати, сводятся многие вышеперечисленные задачи): по имеющемуся признаковому описанию объекта $x \in \mathbb{R}^m$ предсказать значение целевой переменной $y \in \mathbb{R}^k$ для данного объекта. Обычно $k = 1$.

Например, в случае кредитного скоринга x -ом являются все известные о клиенте данные (доход, пол, возраст, кредитная история и т.д.), а y -ом одобрение или неодобрение заявки на кредит.

Библиотеки с алгоритмами машинного обучения, которые будем изучать:

- [scikit-learn](#),
- [XGBoost](#) и
- [pytorch](#).

1.2. Линейная регрессия

Начнем с подключения необходимых библиотек

```
import matplotlib.pyplot as plt
import seaborn as sns
import pandas as pd
import numpy as np
```

Линейная регрессия — это модель следующего вида:

$$a(x) = \langle a, w \rangle + w_0 = \sum_{i=1}^d w_i x_i + w_0, \quad (1)$$

где $w \in \mathbb{R}^d$, $w_0 \in \mathbb{R}$. Параметрами модели являются *веса* или *коэффициенты* w_i . Вес w_0 также называется *свободным коэффициентом* или *сдвигом* (bias). Обучить линейную регрессию — значит найти w и w_0 .

В машинном обучении часто говорят об *обобщающей способности модели*, то есть о способности модели работать на новых, тестовых данных хорошо. Если модель будет идеально предсказывать выборку, на которой она обучалась, но при этом просто ее запомнит, не «вытащив» из данных никакой закономерности, от нее будет мало толку. Такую модель называют *переобученной*: она слишком подстроилась под обучающие примеры, не выявив никакой полезной закономерности, которая позволила бы ей совершать хорошие предсказания на данных, которые она ранее не видела.

Рассмотрим следующий пример, на котором будет хорошо видно, что значит переобучение модели. Для этого нам понадобится сгенерировать синтетические данные. Рассмотрим зависимость $y(x) = \cos(1.5\pi x)$, y — целевая переменная (таргет), а x — объект (просто число от 0 до 1). В жизни мы наблюдаем какое-то конечное количество пар объект-таргет, поэтому смоделируем это, взяв 30 случайных точек x_i в отрезке $[0; 1]$. Более того, в реальной жизни целевая переменная может быть зашумленной (измерения в жизни не всегда точны), смоделируем это, зашумив значение функции нормальным шумом: $\tilde{y}_i = y(x_i) + \mathcal{N}(0, 0.01)$:

```

np.random.seed(36)
x = np.linspace(0, 1, 100)
y = np.cos(1.5 * np.pi * x)

x_objects = np.random.uniform(0, 1, size=30)
y_objects = np.cos(1.5 * np.pi * x_objects) + np.random.normal(scale=0.1, size=x_objects.shape)

```

Попытаемся обучить три разных линейных модели: признаки для первой — $\{x\}$, для второй — $\{x, x^2, x^3, x^4\}$, для третьей — $\{x, \dots, x^{20}\}$:

```

from sklearn.linear_model import LinearRegression
from sklearn.preprocessing import PolynomialFeatures

fig, axs = plt.subplots(figsize=(16, 4), ncols=3)
for i, degree in enumerate([1, 4, 20]):
    X_objects = PolynomialFeatures(degree).fit_transform(x_objects[:, None])
    X = PolynomialFeatures(degree).fit_transform(x[:, None])
    regr = LinearRegression().fit(X_objects, y_objects)
    y_pred = regr.predict(X)
    axs[i].plot(x, y, label="Real function")
    axs[i].scatter(x_objects, y_objects, label="Data")
    axs[i].plot(x, y_pred, label="Prediction")
    if i == 0:
        axs[i].legend()
    axs[i].set_title("Degree = %d" % degree)
    axs[i].set_xlabel("$x$")
    axs[i].set_ylabel("$f(x)$")
    axs[i].set_ylim(-2, 2)

```

Чтобы избежать переобучения, модель регуляризуют. Обычно переобучения в линейных моделях связаны с большими весами, а поэтому модель часто штрафуют за большие значения весов, добавляя к функционалу качества, например, квадрат ℓ^2 -нормы вектора w :

$$Q_{reg}(X, y, a) = Q(X, y, a) + \lambda \|w\|_2^2$$

Это слагаемое называют ℓ_2 -регуляризатором, а коэффициент λ — коэффициентом регуляризации.

1.3. Загрузка данных

Мы будем работать с данными из соревнования [House Prices: Advanced Regression Techniques](#), в котором требовалось предсказать стоимость жилья. Давайте сначала загрузим и немного изучим данные (`train.csv` со страницы соревнования).

```

data = pd.read_csv("train.csv")
data.head()

```

Первое, что стоит заметить — у нас в данных есть уникальное для каждого объекта поле `id`. Обычно такие поля только мешают и способствуют переобучению. Удалим это поле из данных:

```

data = data.drop(columns=["Id"])

```

Разделим данные на обучающую и тестовую выборки. Для простоты не будем выделять дополнительно валидационную выборку (хотя это обычно стоит делать, она нужна для подбора гиперпараметров модели, то есть параметров, которые нельзя подбирать по обучающей выборке). Дополнительно нам придется отделить значения целевой переменной от данных.

```

from sklearn.model_selection import train_test_split

y = data["SalePrice"]
X = data.drop(columns=["SalePrice"])

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=10)

```

Посмотрим сначала на значения целевой переменной:

```
sns.distplot(y_train)
```

Судя по гистограмме, у нас есть примеры с нетипично большой стоимостью, что может помешать нам, если наша функция потерь слишком чувствительна к выбросам. В дальнейшем мы рассмотрим способы, как минимизировать ущерб от этого.

Так как для решения нашей задачи мы бы хотели обучить линейную регрессию, было бы хорошо найти признаки, «наиболее линейно» связанные с целевой переменной, иначе говоря, посмотреть на коэффициент корреляции Пирсона между признаками и целевой переменной. Заметим, что не все признаки являются числовыми, пока что мы не будем рассматривать такие признаки.



Коэффициент корреляции Пирсона

Коэффициент корреляции Пирсона характеризует существование линейной зависимости между двумя величинами.

Пусть даны две выборки $x = (x_1, x_2, \dots, x_m)$ и $y = (y_1, y_2, \dots, y_m)$; коэффициент корреляции Пирсона по формуле:

$$r_{xy} = \frac{\sum_{i=1}^m (x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum_{i=1}^m (x_i - \bar{x})^2 \sum_{i=1}^m (y_i - \bar{y})^2}} = \frac{\text{cov}(x, y)}{\sqrt{s_x^2 s_y^2}},$$

где \bar{x}, \bar{y} — выборочные средние, s_x^2, s_y^2 — выборочные дисперсии, $r_{xy} \in [-1, 1]$.

- $|r_{xy}| = 1 \Rightarrow x, y$ — линейно зависимы,
- $|r_{xy}| = 0 \Rightarrow x, y$ — линейно не зависимы.

```
numeric_data = X_train.select_dtypes([np.number])
numeric_data_mean = numeric_data.mean()
numeric_features = numeric_data.columns

X_train = X_train.fillna(numeric_data_mean)
X_test = X_test.fillna(numeric_data_mean)

correlations = {
    feature: np.corrcoef(X_train[feature], y_train)[0][1]
    for feature in numeric_features
}
sorted_correlations = sorted(correlations.items(), key=lambda x: x[1], reverse=True)
features_order = [x[0] for x in sorted_correlations]
correlations = [x[1] for x in sorted_correlations]

plot = sns.barplot(y=features_order, x=correlations)
plot.figure.set_size_inches(15, 10)
```

Посмотрим на признаки из начала списка. Для этого нарисуем график зависимости целевой переменной от каждого из признаков. На этом графике каждая точка соответствует паре признак-таргет (такие графики называются **scatter-plot**).

```
fig, axs = plt.subplots(figsize=(16, 5), ncols=3)
for i, feature in enumerate(["GrLivArea", "GarageArea", "TotalBsmtSF"]):
    axs[i].scatter(X_train[feature], y_train, alpha=0.2)
    axs[i].set_xlabel(feature)
    axs[i].set_ylabel("SalePrice")
plt.tight_layout()
```

Видим, что между этими признаками и целевой переменной действительно наблюдается линейная зависимость.

1.4. Первая модель

В арсенале дата-саентиста кроме `pandas` и `matplotlib` должны быть библиотеки, позволяющие обучать модели. Для простых моделей (линейные модели, решающее дерево, ...) отлично подходит `sklearn`: в нем очень понятный и простой интерфейс. Несмотря на то, что в `sklearn` есть реализация бустинга и простых нейронных сетей, ими все же не пользуются и предпочитают специализированные библиотеки: `XGBoost`, `LightGBM` и пр. для градиентного бустинга над деревьями, `PyTorch`, `Tensorflow` и пр. для нейронных сетей. Так как мы будем обучать линейную регрессию, нам подойдет реализация из `sklearn`.

Попробуем обучить линейную регрессию на числовых признаках из нашего датасета. В `sklearn` есть несколько классов, реализующих линейную регрессию:

- **LinearRegression** — «классическая» линейная регрессия с оптимизацией MSE. Веса находятся как точное решение: $w^* = (X^T X)^{-1} X^T y$
- **Ridge** — линейная регрессия с оптимизацией MSE и ℓ_2 -регуляризацией
- **Lasso** — линейная регрессия с оптимизацией MSE и ℓ_1 -регуляризацией

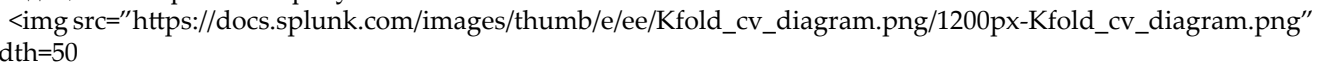
У моделей из `sklearn` есть методы `fit` и `predict`. Первый принимает на вход обучающую выборку и вектор целевых переменных и обучает модель, второй, будучи вызванным после обучения модели, возвращает предсказание на выборке. Попробуем обучить нашу первую модель на числовых признаках, которые у нас сейчас есть:

```
from sklearn.linear_model import Ridge
from sklearn.metrics import mean_squared_error

model = Ridge()
model.fit(X_train[numeric_features], y_train)
y_pred = model.predict(X_test[numeric_features])
y_train_pred = model.predict(X_train[numeric_features])

print("Test MSE = %.4f" % mean_squared_error(y_test, y_pred))
print("Train MSE = %.4f" % mean_squared_error(y_train, y_train_pred))
```

Мы обучили первую модель и даже посчитали ее качество на отложенной выборке! Давайте теперь посмотрим на то, как можно оценить качество модели с помощью кросс-валидации. Принцип кросс-валидации изображен на рисунке

width=50

При кросс-валидации мы делим обучающую выборку на n частей (fold). Затем мы обучаем n моделей: каждая модель обучается при отсутствии соответствующего фолда, то есть i -ая модель обучается на всей обучающей выборке, кроме объектов, которые попали в i -ый фолд (out-of-fold). Затем мы измеряем качество i -ой модели на i -ом фолде. Так как он не участвовал в обучении этой модели, мы получим «честный результат». После этого, для получения финального значения метрики качества, мы можем усреднить полученные нами n значений.

```
from sklearn.model_selection import cross_val_score

cv_scores = cross_val_score(model, X_train[numeric_features], y_train, cv=10, scoring="neg_mean_squared_error")
print("Cross validation scores:\n\t", "\n\t".join("%.4f" % x for x in cv_scores))
print("Mean CV MSE = %.4f" % np.mean(-cv_scores))
```

Обратите внимание на то, что результаты `cv_scores` получились отрицательными. Это соглашение в `sklearn` (скоринговую функцию нужно максимизировать). Поэтому все стандартные скореры называются `neg_*`, например, `neg_mean_squared_error`.

В качестве метрики качества в соревновании использовалось RMSE (Root Mean Squared Error), а не MSE, которое мы считали выше (и по отложенной выборке и при кросс-валидации):

$$\text{RMSE}(X, y, a) = \sqrt{\frac{1}{\ell} \sum_{i=1}^{\ell} (y_i - a(x_i))^2}$$

RMSE в чистом виде не входит в стандартные метрики `sklearn`, но мы всегда можем определить свою метрику и использовать ее в некоторых функциях `sklearn`, например, `cross_val_score`. Для этого нужно воспользоваться `sklearn.metrics.make_scorer`.

```
from sklearn.metrics import make_scorer
```

```
def rmse(y_true, y_pred):  
    error = (y_true - y_pred) ** 2  
    return np.sqrt(np.mean(error))
```

```
rmse_scorer = make_scorer(  
    rmse,  
    greater_is_better=False  
)
```

```
from sklearn.linear_model import Ridge
```

```
model = Ridge()  
model.fit(X_train[numeric_features], y_train)  
y_pred = model.predict(X_test[numeric_features])  
y_train_pred = model.predict(X_train[numeric_features])  
  
print("Test RMSE = %.4f" % rmse(y_test, y_pred))  
print("Train RMSE = %.4f" % rmse(y_train, y_train_pred))
```

```
from sklearn.model_selection import cross_val_score
```

```
cv_scores = cross_val_score(model, X_train[numeric_features], y_train, cv=10, scoring=rmse_scorer)  
print("Cross validation scores:\n\t", "\n\t".join("%.4f" % x for x in cv_scores))  
print("Mean CV RMSE = %.4f" % np.mean(-cv_scores))
```

Для того, чтобы иметь некоторую точку отсчета, удобно посчитать оптимальное значение функции потерь при константном предсказании.

```
best_constant = y_train.mean()  
print("Test RMSE with best constant = %.4f" % rmse(y_test, best_constant))  
print("Train RMSE with best constant = %.4f" % rmse(y_train, best_constant))
```

Давайте посмотрим на то, какие же признаки оказались самыми «сильными». Для этого визуализируем веса, соответствующие признакам. Чем больше вес — тем более сильным является признак.

```
def show_weights(features, weights, scales):  
    fig, axs = plt.subplots(figsize=(14, 10), ncols=2)  
    sorted_weights = sorted(zip(weights, features, scales), reverse=True)  
    weights = [x[0] for x in sorted_weights]  
    features = [x[1] for x in sorted_weights]  
    scales = [x[2] for x in sorted_weights]  
    sns.barplot(y=features, x=weights, ax=axs[0])  
    axs[0].set_xlabel("Weight")  
    sns.barplot(y=features, x=scales, ax=axs[1])  
    axs[1].set_xlabel("Scale")  
    plt.tight_layout()
```

```
show_weights(numeric_features, model.coef_, X_train[numeric_features].std())
```

Будем масштабировать наши признаки перед обучением модели. Это, среди прочего, сделает нашу регуляризацию более честной: теперь все признаки будут регуляризоваться в равной степени.

Для этого воспользуемся трансформером [StandardScaler](#). Трансформеры в `sklearn` имеют методы `fit` и `transform` (а еще `fit_transform`). Метод `fit` принимает на вход обучающую выборку и считает по ней необходимые значения (например статистики, как `StandardScaler`: среднее и стандартное отклонение каждого из признаков); `transform` применяет преобразование к переданной выборке.

```
from sklearn.preprocessing import StandardScaler
```

```
scaler = StandardScaler()
```

```

X_train_scaled = scaler.fit_transform(X_train[numeric_features])
X_test_scaled = scaler.transform(X_test[numeric_features])

model = Ridge()
model.fit(X_train_scaled, y_train)
y_pred = model.predict(X_test_scaled)
y_train_pred = model.predict(X_train_scaled)

print("Test RMSE = %.4f" % rmse(y_test, y_pred))
print("Train RMSE = %.4f" % rmse(y_train, y_train_pred))

```

```

scales = pd.Series(data=X_train_scaled.std(axis=0), index=numeric_features)
show_weights(numeric_features, model.coef_, scales)

```

Наряду с параметрами (веса w , w_0), которые модель оптимизирует на этапе обучения, у модели есть и гиперпараметры. У нашей модели это **alpha** — коэффициент регуляризации. Подбирают его обычно по сетке, измеряя качество на валидационной (не тестовой) выборке или с помощью кросс-валидации. Посмотрим, как это можно сделать (заметьте, что мы перебираем **alpha** по логарифмической сетке, чтобы узнать оптимальный порядок величины).

```

from sklearn.model_selection import GridSearchCV

alphas = np.logspace(-2, 3, 20)
searcher = GridSearchCV(Ridge(), [{"alpha": alphas}], scoring=rmse_scorer, cv=10)
searcher.fit(X_train_scaled, y_train)

best_alpha = searcher.best_params_["alpha"]
print("Best alpha = %.4f" % best_alpha)

plt.plot(alphas, -searcher.cv_results_["mean_test_score"])
plt.xscale("log")
plt.xlabel("alpha")
plt.ylabel("CV score")

```

Попробуем обучить модель с подобранным коэффициентом регуляризации. Заодно воспользуемся очень удобным классом **Pipeline**: обучение модели часто представляется как последовательность некоторых действий с обучающей и тестовой выборками (например, сначала нужно отмасштабировать выборку (причем для обучающей выборки нужно применить метод **fit**, а для тестовой — **transform**), а затем обучить/применить модель (для обучающей **fit**, а для тестовой — **predict**). **Pipeline** позволяет хранить эту последовательность шагов и корректно обрабатывает разные типы выборок: и обучающую, и тестовую.

```

from sklearn.pipeline import Pipeline

simple_pipeline = Pipeline([
    ('scaling', StandardScaler()),
    ('regression', Ridge(best_alpha))
])

model = simple_pipeline.fit(X_train[numeric_features], y_train)
y_pred = model.predict(X_test[numeric_features])
print("Test RMSE = %.4f" % rmse(y_test, y_pred))

```

1.5. Работа с категориальными признаками

Сейчас мы явно вытягиваем из данных не всю информацию, что у нас есть, просто потому, что мы не используем часть признаков. Эти признаки в датасете закодированы строками, каждый из них обозначает некоторую категорию. Такие признаки называются категориальными. Давайте выделим такие признаки и сразу заполним пропуски в них специальным значением (то, что у признака пропущено значение, само по себе может быть хорошим признаком).

```

categorical = list(X_train.dtypes[X_train.dtypes == "object"].index)
X_train[categorical] = X_train[categorical].fillna("NotGiven")
X_test[categorical] = X_test[categorical].fillna("NotGiven")

```

```
X_train[categorical].sample(5)
```

Сейчас нам нужно как-то закодировать эти категориальные признаки числами, ведь линейная модель не может работать с такими абстракциями. Два стандартных трансформера из `sklearn` для работы с категориальными признаками

- `LabelEncoder` просто перенумеровывает значения признака натуральными числами
- `OneHotEncoder` ставит в соответствие каждому признаку целый вектор, состоящий из нулей и одной единицы (которая стоит на месте, соответствующем принимаемому значению, таким образом кодируя его).

```
from sklearn.preprocessing import OneHotEncoder
from sklearn.compose import ColumnTransformer

column_transformer = ColumnTransformer([
    ('ohe', OneHotEncoder(handle_unknown="ignore"), categorical),
    ('scaling', StandardScaler(), numeric_features)
])

pipeline = Pipeline(steps=[
    ('ohe_and_scaling', column_transformer),
    ('regression', Ridge())
])

model = pipeline.fit(X_train, y_train)
y_pred = model.predict(X_test)
print("Test RMSE = %.4f" % rmse(y_test, y_pred))
```

Посмотрим на размеры матрицы после OneHot-кодирования:

```
print("Size before OneHot:", X_train.shape)
print("Size after OneHot:", column_transformer.transform(X_train).shape)
```

Как видим, количество признаков увеличилось более, чем в 3 раза. Это может повысить риски переобучиться: соотношение количества объектов к количеству признаков сильно сократилось.

Попытаемся обучить линейную регрессию с ℓ_1 -регуляризатором. На лекциях вы узнаете, что ℓ_1 -регуляризатор разреживает признаковое пространство, иными словами, такая модель зануляет часть весов.

```
from sklearn.linear_model import Lasso

column_transformer = ColumnTransformer([
    ('ohe', OneHotEncoder(handle_unknown="ignore"), categorical),
    ('scaling', StandardScaler(), numeric_features)
])

lasso_pipeline = Pipeline(steps=[
    ('ohe_and_scaling', column_transformer),
    ('regression', Lasso())
])

model = lasso_pipeline.fit(X_train, y_train)
y_pred = model.predict(X_test)
print("RMSE = %.4f" % rmse(y_test, y_pred))

ridge_zeros = np.sum(pipeline.steps[-1][-1].coef_ == 0)
lasso_zeros = np.sum(lasso_pipeline.steps[-1][-1].coef_ == 0)
print("Zero weights in Ridge:", ridge_zeros)
print("Zero weights in Lasso:", lasso_zeros)
```

Подберем для нашей модели оптимальный коэффициент регуляризации. Обратите внимание, как перебираются параметры у `Pipeline`.

```

alphas = np.logspace(-2, 4, 20)
searcher = GridSearchCV(lasso_pipeline, [{"regression__alpha": alphas}], scoring=rmse_scorer, cv=10)
searcher.fit(X_train, y_train)

best_alpha = searcher.best_params_["regression__alpha"]
print("Best alpha = %.4f" % best_alpha)

plt.plot(alphas, -searcher.cv_results_["mean_test_score"])
plt.xscale("log")
plt.xlabel("alpha")
plt.ylabel("CV score")

```

```

column_transformer = ColumnTransformer([
    ('ohe', OneHotEncoder(handle_unknown="ignore"), categorical),
    ('scaling', StandardScaler(), numeric_features)
])

pipeline = Pipeline(steps=[
    ('ohe_and_scaling', column_transformer),
    ('regression', Lasso(best_alpha))
])

model = pipeline.fit(X_train, y_train)
y_pred = model.predict(X_test)
print("Test RMSE = %.4f" % rmse(y_test, y_pred))

```

```

lasso_zeros = np.sum(pipeline.steps[-1][-1].coef_ == 0)
print("Zero weights in Lasso:", lasso_zeros)

```

Иногда очень полезно посмотреть на распределение остатков. Нарисуем гистограмму распределения квадратичной ошибки на обучающих объектах:

```

error = (y_train - model.predict(X_train)) ** 2
sns.distplot(error)

```

Как видно из гистограммы, есть примеры с очень большими остатками. Попробуем их выбросить из обучающей выборки. Например, выбросим примеры, остаток у которых больше 0.95-квантили.

```

mask = (error < np.quantile(error, 0.95))

```

```

column_transformer = ColumnTransformer([
    ('ohe', OneHotEncoder(handle_unknown="ignore"), categorical),
    ('scaling', StandardScaler(), numeric_features)
])

pipeline = Pipeline(steps=[
    ('ohe_and_scaling', column_transformer),
    ('regression', Lasso(best_alpha))
])

model = pipeline.fit(X_train[mask], y_train[mask])
y_pred = model.predict(X_test)
print("Test RMSE = %.4f" % rmse(y_test, y_pred))

```

```

X_train = X_train[mask]
y_train = y_train[mask]

```

```

error = (y_train - model.predict(X_train)) ** 2
sns.distplot(error)

```

Видим, что качество модели заметно улучшилось! Также бывает очень полезно посмотреть на примеры с большими остатками и попытаться понять, почему же модель на них так сильно ошибается: это может дать понимание, как модель можно улучшить.

2. Предобработка данных

Начнем с подключения необходимых библиотек и модулей:

```
import pandas as pd
import seaborn as sns
from tqdm import tqdm
from sklearn.datasets import fetch_20newsgroups

from sklearn.model_selection import train_test_split
from sklearn.linear_model import Ridge
from sklearn.metrics import mean_squared_error
```

2.1. Работа с текстовыми данными

Как правило, модели машинного обучения действуют в предположении, что матрица «объект-признак» является вещественнозначной, поэтому при работе с текстами сперва для каждого из них необходимо составить его признаковое описание. Для этого широко используются техники векторизации, tf-idf и пр.

Сперва загрузим данные:

```
data = fetch_20newsgroups(subset='all', categories=['comp.graphics', 'sci.med'])
```

Данные содержат тексты новостей, которые надо классифицировать на разделы.

```
data['target_names']
```

```
texts = data['data']
target = data['target']
```

Например:

```
texts[0]
```

```
data['target_names'][target[0]]
```

Bag-of-words. Самый очевидный способ формирования признакового описания текстов — векторизация. Простой способ заключается в подсчёте, сколько раз встретилось каждое слово в тексте. Получаем вектор длиной в количество уникальных слов, встречающихся во всех объектах выборки. В таком векторе много нулей, поэтому его удобнее хранить в разреженном виде.

Пусть у нас имеется коллекция текстов $D = \{d_i\}_{i=1}^l$ и словарь всех слов, встречающихся в выборке $V = \{v_j\}_{j=1}^d$. В этом случае некоторый текст d_i описывается вектором $(x_{ij})_{j=1}^d$, где

$$x_{ij} = \sum_{v \in d_i} [v = v_j].$$

Таким образом, текст d_i описывается вектором количества вхождений каждого слова из словаря в данный текст.

```
from sklearn.feature_extraction.text import CountVectorizer

vectorizer = CountVectorizer(encoding='utf8', min_df=1)
_ = vectorizer.fit(texts)
```

Результатом является разреженная матрица.

```
vectorizer.transform(texts[:1])
```

```
print(vectorizer.transform(texts[:1]).indptr)
print(vectorizer.transform(texts[:1]).indices)
print(vectorizer.transform(texts[:1]).data)
```

Такой способ представления текстов называют *мешком слов* (bag-of-words).

TF-IDF. Очевидно, что не все слова полезны в задаче прогнозирования. Например, мало информации несут слова, встречающиеся во всех текстах. Это могут быть как стоп-слова, так и слова, свойственные всем текстам выборки (в текстах про автомобили употребляется слово «автомобиль»). Эту проблему решает TF-IDF (*T*erm *F*requency–*I*nverse *D*ocument *F*requency) преобразование текста.

Рассмотрим коллекцию текстов D . Для каждого уникального слова t из документа $d \in D$ вычислим следующие величины:

- TD (Term Frequency) – количество вхождений слова в отношении к общему числу слов в тексте:

$$tf(t, d) = \frac{n_{td}}{\sum_{t \in d} n_{td}},$$

где n_{td} — количество вхождений слова t в текст d .

- IDF (Inverse Document Frequency):

$$idf(t, D) = \log \frac{|D|}{|\{d \in D : t \in d\}|},$$

где $|\{d \in D : t \in d\}|$ – количество текстов в коллекции, содержащих слово t .

Тогда для каждой пары (слово, текст) (t, d) вычислим величину:

$$tf-idf(t, d, D) = tf(t, d) \cdot idf(t, D).$$

Отметим, что значение $tf(t, d)$ корректируется для часто встречающихся общеупотребимых слов при помощи значения $idf(t, D)$.

```
from sklearn.feature_extraction.text import TfidfVectorizer

vectorizer = TfidfVectorizer(encoding='utf8', min_df=1)
_ = vectorizer.fit(texts)
```

На выходе получаем разреженную матрицу.

```
vectorizer.transform(texts[:1])
```

```
print(vectorizer.transform(texts[:1]).indptr)
print(vectorizer.transform(texts[:1]).indices)
print(vectorizer.transform(texts[:1]).data)
```

Заметим, что оба метода возвращают вектор длины 32548 (размер нашего словаря).

Заметим, что одно и то же слово может встречаться в различных формах (например, «сотрудник» и «сотрудника»), но описанные выше методы интерпретируют их как различные слова, что делает признаковое описание избыточным. Устранить эту проблему можно при помощи **лемматизации** и **стемминга**.

Стемминг. *Стемминг* — это процесс нахождения основы слова. В результате применения данной процедуры однокоренные слова, как правило, преобразуются к одинаковому виду.

Таблица 1: Примеры стемминга

| Слово | Основа |
|------------|---------|
| вагон | вагон |
| вагона | вагон |
| вагоне | вагон |
| вагонов | вагон |
| вагоном | вагон |
| вагоны | вагон |
| важная | важн |
| важнее | важн |
| важнейшие | важн |
| важнейшими | важн |
| важничал | важнича |
| важно | важн |

Snowball — фреймворк для написания алгоритмов стемминга (библиотека `nltk`). Алгоритмы стемминга отличаются для разных языков и используют знания о конкретном языке — списки окончаний для разных чистей речи, разных склонений и т.д. Пример алгоритма для русского языка — [Russian stemming](#).

```
import nltk
stemmer = nltk.stem.snowball.RussianStemmer()

print(stemmer.stem('машинное'), stemmer.stem('обучение'))

stemmer = nltk.stem.snowball.EnglishStemmer()

def stem_text(text, stemmer):
    tokens = text.split()
    return ' '.join(map(lambda w: stemmer.stem(w), tokens))

stemmed_texts = []
for t in tqdm(texts[:1000]):
    stemmed_texts.append(stem_text(t, stemmer))

print(texts[0])

print(stemmed_texts[0])
```

Как видим, стеммер работает не очень быстро и запускать его для всей выборки достаточно накладно.

Лемматизация. *Лемматизация* — процесс приведения слова к его нормальной форме (лемме):

- для существительных — именительный падеж, единственное число;
- для прилагательных — именительный падеж, единственное число, мужской род;
- для глаголов, причастий, деепричастий — глагол в инфинитиве.

Лемматизация — процесс более сложный по сравнению со стеммингом. Стеммер просто «режет» слово до основы.

Например, для русского языка есть библиотека `pymorphy2`.

```
import pymorphy2
morph = pymorphy2.MorphAnalyzer()
```

```
morph.parse('играющих')[0]
```

Сравним работу стеммера и лемматизатора на примере:

```
stemmer = nltk.stem.snowball.RussianStemmer()
print(stemmer.stem('играющих'))
```

```
print(morph.parse('играющих')[0].normal_form)
```

2.2. Трансформация признаков и целевой переменной

Разберёмся, как может влиять трансформация признаков или целевой переменной на качество модели.

Логарифмирование. Воспользуемся датасетом с ценами на дома, с которым мы уже сталкивались ранее ([House Prices: Advanced Regression Techniques](#)).

```
data = pd.read_csv('train.csv')

data = data.drop(columns=['Id'])
y = data['SalePrice']
X = data.drop(columns=['SalePrice'])
```

Посмотрим на распределение целевой переменной

```
plt.figure(figsize=(12, 5))

plt.subplot(1, 2, 1)
sns.distplot(y, label='target')
plt.title('target')

plt.subplot(1, 2, 2)
sns.distplot(data.GrLivArea, label='area')
plt.title('area')
plt.show()
```

Видим, что распределения несимметричны с тяжёлыми правыми хвостами. Оставим только числовые признаки, пропуски заменим средним значением.

```
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.3, random_state=10)

numeric_data = X_train.select_dtypes([np.number])
numeric_data_mean = numeric_data.mean()
numeric_features = numeric_data.columns

X_train = X_train.fillna(numeric_data_mean)[numeric_features]
X_test = X_test.fillna(numeric_data_mean)[numeric_features]
```

Если разбирать линейную регрессию с вероятностной точки зрения, то можно получить, что шум должен быть распределён нормально. Поэтому лучше, когда целевая переменная распределена также нормально.

Если прологарифмировать целевую переменную, то её распределение станет больше похоже на нормальное:

```
sns.distplot(np.log(y+1), label='target')
plt.show()
```

Сравним качество линейной регрессии в двух случаях:

- Целевая переменная без изменений.
- Целевая переменная прологарифмирована.



Предупреждение

Не забудем во втором случае взять экспоненту от предсказаний!

```
model = Ridge()
model.fit(X_train, y_train)
y_pred = model.predict(X_test)

print("Test RMSE = %.4f" % mean_squared_error(y_test, y_pred) ** 0.5)
```

```
model = Ridge()
model.fit(X_train, np.log(y_train+1))
y_pred = np.exp(model.predict(X_test))-1

print("Test RMSE = %.4f" % mean_squared_error(y_test, y_pred) ** 0.5)
```

Попробуем аналогично логарифмировать один из признаков, имеющих также смещённое распределение (этот признак был вторым по важности!)

```
X_train.GrLivArea = np.log(X_train.GrLivArea + 1)
X_test.GrLivArea = np.log(X_test.GrLivArea + 1)
```

```
model = Ridge()
model.fit(X_train[numeric_features], y_train)
y_pred = model.predict(X_test[numeric_features])

print("Test RMSE = %.4f" % mean_squared_error(y_test, y_pred) ** 0.5)
```

```
model = Ridge()
model.fit(X_train[numeric_features], np.log(y_train+1))
y_pred = np.exp(model.predict(X_test[numeric_features]))-1

print("Test RMSE = %.4f" % mean_squared_error(y_test, y_pred) ** 0.5)
```

Как видим, преобразование признаков влияет слабее. Признаков много, а вклад размывается по всем. К тому же, проверять распределение множества признаков технически сложнее, чем одной целевой переменной.

2.3. Бинаризация

Мы уже смотрели, как полиномиальные признаки могут помочь при восстановлении нелинейной зависимости линейной моделью. Альтернативный подход заключается в бинаризации признаков. Мы разбиваем ось значений одного из признаков на куски (бины) и добавляем для каждого куска-бина новый признак-индикатор попадания в этот бин.

```
from sklearn.linear_model import LinearRegression

np.random.seed(36)
X = np.random.uniform(0, 1, size=100)
y = np.cos(1.5 * np.pi * X) + np.random.normal(scale=0.1, size=X.shape)

plt.scatter(X, y)
```

```
X = X.reshape((-1, 1))
thresholds = np.arange(0.2, 1.1, 0.2).reshape((1, -1))

X_expand = np.hstack((
    X,
    ((X > thresholds[:, :-1]) & (X <= thresholds[:, 1:])).astype(int)))
```

```
from sklearn.model_selection import KFold
from sklearn.model_selection import cross_val_score
```

```
-np.mean(cross_val_score(
    LinearRegression(), X, y, cv=KFold(n_splits=3, random_state=123),
    scoring='neg_mean_squared_error'))
```

```
-np.mean(cross_val_score(
    LinearRegression(), X_expand, y, cv=KFold(n_splits=3, random_state=123),
    scoring='neg_mean_squared_error'))
```

Так линейная модель может лучше восстанавливать нелинейные зависимости.

2.4. Транзакционные данные

Напоследок посмотрим, как можно извлекать признаки из транзакционных данных.

Транзакционные данные характеризуются тем, что есть много строк, характеризующихся моментами времени и некоторым числом (суммой денег, например). При этом если это банк, то каждому человеку принадлежит не одна транзакция, а чаще всего надо предсказывать некоторые сущности для клиентов. Таким образом, надо получить признаки для пользователей из множества их транзакций. Этим мы и займёмся.

Для примера возьмём данные [отсюда](#). Задача детектирования фродовых клиентов.

```
customers = pd.read_csv('Retail_Data_Response.csv')
transactions = pd.read_csv('Retail_Data_Transactions.csv')
```

```
customers.head()
```

```
transactions.head()
```

```
transactions.trans_date = transactions.trans_date.apply(
    lambda x: datetime.datetime.strptime(x, '%d-%b-%y'))
```

Посмотрим на распределение целевой переменной:

```
customers.response.mean()
```

Получаем примерно 1 к 9 положительных примеров. Если такие данные разбивать на части для кросс-валидации, то может получиться так, что в одну из частей попадёт слишком мало положительных примеров, а в другую — наоборот. На случай такого неравномерного баланса классов есть `StratifiedKFold`, который бьёт данные так, чтобы баланс классов во всех частях был одинаковым.

```
from sklearn.model_selection import StratifiedKFold
```

Когда строк на каждый объект много, можно считать различные статистики. Например, средние, минимальные и максимальные суммы, потраченные клиентом, количество транзакций, ...

```
agg_transactions = transactions.groupby('customer_id').tran_amount.agg(
    ['mean', 'std', 'count', 'min', 'max']).reset_index()

data = pd.merge(customers, agg_transactions, how='left', on='customer_id')

data.head()
```

```
from sklearn.linear_model import LogisticRegression

np.mean(cross_val_score(
    LogisticRegression(),
    X=data.drop(['customer_id', 'response'], axis=1),
    y=data.response,
    cv=StratifiedKFold(n_splits=3, random_state=123),
    scoring='roc_auc'))
```

Но каждая транзакция снабжена датой! Можно посчитать статистики только по свежим транзакциям. Добавим их.

```
transactions.trans_date.min(), transactions.trans_date.max()
```

```
agg_transactions = transactions.loc[transactions.trans_date.apply(
    lambda x: x.year == 2014)].groupby('customer_id').tran_amount.agg(
    ['mean', 'std', 'count', 'min', 'max']).reset_index()
```

```
data = pd.merge(data, agg_transactions, how='left', on='customer_id', suffixes=('', '_2014'))
data = data.fillna(0)
```

```
np.mean(cross_val_score(
    LogisticRegression(),
    X=data.drop(['customer_id', 'response'], axis=1),
    y=data.response,
    cv=StratifiedKFold(n_splits=3, random_state=123),
    scoring='roc_auc'))
```

Можно также считать дату первой и последней транзакциями пользователей, среднее время между транзакциями и прочее.

3. Простые модели классификации

Классификация — отнесение объекта к одной из категорий на основании его признаков.

Рассмотрим задачу бинарной классификации. Пусть $X = \mathbb{R}^d$ — пространство объектов, $Y = 1, +1$ — множество допустимых ответов, $X = (x_i, y_i)_{i=1}^{\ell}$ — обучающая выборка. Иногда мы будем класс «+1» называть положительным, а класс «-1» — отрицательным.

Будем считать, что классификатор имеет вид

$$a(x) = \text{sign}(b(x)t) = 2[b(x) > t]1.$$

В такого рода задачах возникает необходимость в изучении различных аспектов качества уже обученного классификатора. Сначала обсудим один из подходов к измерению качества таких моделей.

3.1. Матрица ошибок

Матрица ошибок — это способ разбить объекты на четыре категории в зависимости от комбинации истинного ответа и ответа алгоритма (см. таблицу 2). Через элементы этой матрицы можно, например, выразить долю правильных ответов:

$$\text{accuracy} = \frac{\text{TP} + \text{TN}}{\text{TP} + \text{FP} + \text{FN} + \text{TN}}.$$

Таблица 2: Матрица ошибок

| $y = 1$ | $y = -1$ |
|---------------------|---------------------|
| TP (True Positive) | FP (False Positive) |
| FN (False Negative) | TN (True Negative) |

Данная матрица имеет существенный недостаток — её значение необходимо оценивать в контексте баланса классов. Если в выборке 950 отрицательных и 50 положительных объектов, то при абсолютно случайной классификации мы получим долю правильных ответов 0.95. Это означает, что доля положительных ответов сама по себе не несет никакой информации о качестве работы алгоритма $a(x)$, и вместе с ней следует анализировать соотношение классов в выборке.

Гораздо более информативными критериями являются *точность* (precision) и *полнота* (recall).

Точность показывает, какая доля объектов, выделенных классификатором как положительные, действительно является положительными:

$$\text{precision} = \frac{\text{TP}}{\text{TP} + \text{FP}}$$

Полнота показывает, какая часть положительных объектов была выделена классификатором:

$$\text{recall} = \frac{\text{TP}}{\text{TP} + \text{FN}}$$

Существует несколько способов получить один критерий качества на основе точности и полноты. Один из них — F -мера, гармоническое среднее точности и полноты:

$$F_\beta = (1 + \beta^2) \frac{\text{precision} \cdot \text{recall}}{\beta^2 \cdot \text{precision} + \text{recall}}.$$

Среднее гармоническое обладает важным свойством — оно близко к нулю, если хотя бы один из аргументов близок к нулю. Именно поэтому оно является более предпочтительным, чем среднее арифметическое (если алгоритм будет относить все объекты к положительному классу, то он будет иметь $\text{recall} = 1$ и $\text{precision} > 0$, а их среднее арифметическое будет больше $1/2$, что недопустимо).

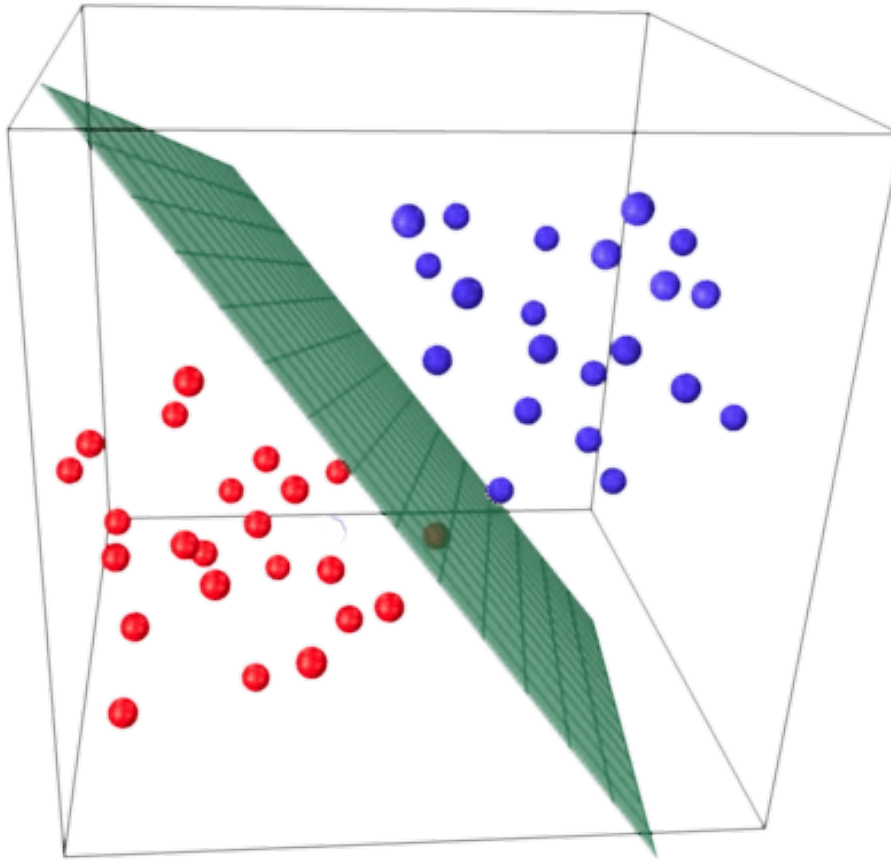
Чаще всего берут $\beta = 1$ хотя иногда встречаются и другие модификации. F_2 острее реагирует на recall (т. е. на долю ложноположительных ответов), а $F_{0.5}$ чувствительнее к точности (ослабляет влияние ложноположительных ответов).

В `sklearn` есть удобная функция `sklearn.metrics.classification_report`, которая возвращает recall , precision и F -меру для каждого из классов, а также количество экземпляров каждого класса.

```
from sklearn.metrics import classification_report
y_true = [0, 1, 2, 2, 2]
y_pred = [0, 0, 2, 2, 1]
target_names = ['class 0', 'class 1', 'class 2']
print(classification_report(y_true, y_pred, target_names=target_names))
```

3.2. Линейная классификация

Основная идея линейного классификатора заключается в том, что признаковое пространство может быть разделено гиперплоскостью на две полуплоскости, в каждой из которых прогнозируется одно из двух значений целевого класса. Если это можно сделать без ошибок, то обучающая выборка называется *линейно разделимой*.



Указанная разделяющая плоскость называется *линейным дискриминантом*.

Логистическая регрессия. Логистическая регрессия является частным случаем линейного классификатора, но она обладает хорошим «умением» – прогнозировать вероятность отнесения наблюдения к классу. Таким образом, результат логистической регрессии всегда находится на отрезке $[0, 1]$. Возьмем данные по *ирисам*

```
iris = pd.read_csv("https://raw.githubusercontent.com/mwaskom/seaborn-data/master/iris.csv")
```

```
iris.describe()
```

```
sns.pairplot(iris, hue="species")
```

```
sns.lmplot(x="petal_length", y="petal_width", data=iris)
```

```
X = iris.iloc[:, 2:4].values
y = iris['species'].values
```

```
y[:5]
```

```
from sklearn.preprocessing import LabelEncoder
```

```
le = LabelEncoder()
le.fit(y)
y = le.transform(y)
y[:5]
```

```
iris_pred_names = le.classes_  
iris_pred_names
```

```
from sklearn.model_selection import train_test_split  
  
X_train, X_test, y_train, y_test = train_test_split(  
    X, y, test_size=0.3, random_state=0)
```

```
from sklearn.preprocessing import StandardScaler  
  
sc = StandardScaler()  
sc.fit(X_train)  
X_train_std = sc.transform(X_train)  
X_test_std = sc.transform(X_test)
```

```
X_train[:5], X_train_std[:5]
```

```
from sklearn.linear_model import LogisticRegression  
  
lr = LogisticRegression(C=100.0, random_state=1)  
lr.fit(X_train_std, y_train)
```

```
lr.predict_proba(X_test_std[:3, :])
```

```
lr.predict_proba(X_test_std[:3, :]).sum(axis=1)
```

```
y_test[:3]
```

```
lr.predict_proba(X_test_std[:3, :]).argmax(axis=1)
```

Предсказываем класс первого наблюдения

```
lr.predict(X_test_std[0, :].reshape(1, -1))
```

На основе его коэффициентов:

```
X_test_std[0, :]
```

```
X_test_std[0, :].reshape(1, -1)
```

```
y_pred = lr.predict(X_test_std)
```

```
len(iris_pred_names)
```

```
print(classification_report(y_test, y_pred, target_names=iris_pred_names))
```

4. Задание

Задание состоит из двух основных частей. В первой части необходимо сделать простой препроцессинг и произвести разведывательный анализ данных.

Во второй части у Вас будет выбор между двумя вариантами: Вы можете провести регрессионный анализ данных или заняться обработкой естественного языка и построением классификатора текстов.

4.1. Задание по базе wine

a)

Загрузка и разведывательный анализ.

- Загрузите данные ([скачать](#)).
- Посчитайте размерность данных.
- Посчитайте количество пропущенных значений в каждой переменной.
- Выведите тип данных каждой переменной. Переконвертируйте при необходимости.
- Вина какой области (province) получают наилучшие рейтинги?
- На основе словаря color создайте переменную, в которой закодирован цвет вина.
- Удалите наблюдения для которых цвет (color) не указан.
- Визуализируйте распределения числовых переменных.
- Для каждой страны рассчитайте долю каждого вида вина. В какой стране доля белого вина наибольшая, а в какой красного? (Нужен ответ вида: в стране А наибольшая доля белого вина, а в стране В — красного).
- Разделите выборку на обучающую и тестовую

b)

Регрессионная модель.

- На обучающей выборке постройте регрессионную модель, показывающую зависимость между баллом (зависимая переменная) и ценой. Визуализируйте эту зависимость. На сколько изменится оценка при изменении цены на одну условную единицу?
- Оцените качество модели на основе предсказаний по тестовой выборке по помощи стандартных метрик качества для регрессионных моделей.
- Добавьте в модель переменную, в которой закодирован цвет вина. Как изменилось качество?

ИЛИ

c)

Классификация текстов.

- Сделайте препроцессинг текстов в поле description.
- На обучающей выборке постройте модель классификации текста, которая бы классифицировала вина по цвету на основе текстов из описания.
- Оцените качество работы модели по помощи стандартных метрик качества для алгоритмов классификации. Использование автоматических методов подбора параметров (Grid Search) не обязательно, но в случае наличия — зачтётся.

Имя файла: `task_surname.ipynb`.