Машинное обучение

Пензин М.С.

penzin.ml.tsu@gmail.com (mailto:penzin.ml.tsu@gmail.com)

In [3]:
```
%matplotlib inline
import numpy as np
import pandas as pd
import seaborn as sns
from matplotlib import pyplot as plt
import warnings
warnings.filterwarnings('ignore')
from pylab import rcParams
```

In [4]:
```
def AMS(w, y, y_pred):
    """
        Расчет метрики, работает только с Numpy-массивами
    """
    s = (w * (y == 1) * (y_pred == 1)).sum()
    b = (w * (y == 0) * (y_pred == 1)).sum()
    bReg = 10.
    return np.sqrt(2 * ((s + b + bReg) *
```

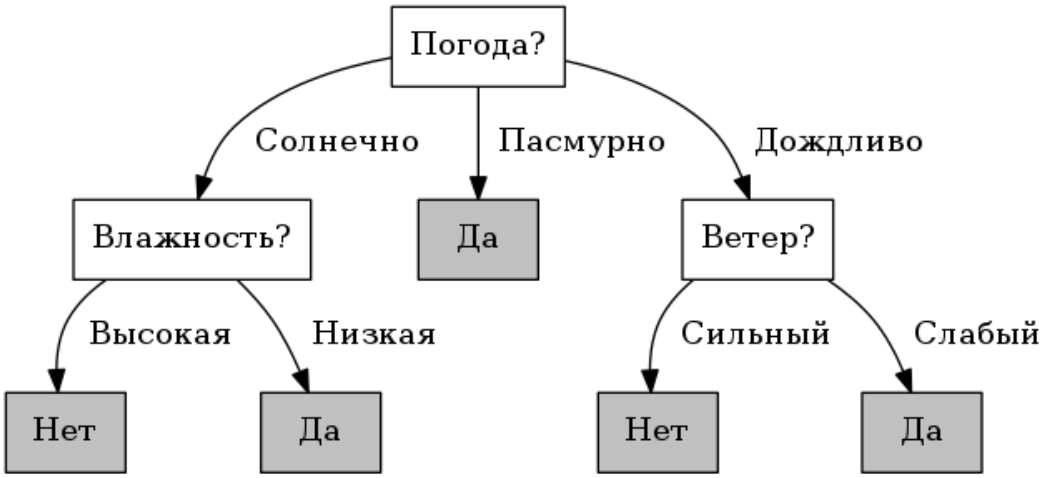
In [10]:
```
## Долгая скучная инициализация данных

def loadTrain():
    import zipfile
    z = zipfile.ZipFile("data/training.zip")
    df = pd.read_csv(z.open("training.csv"))
    return df

df = loadTrain()

df[["log_PRI_met_sumet", "log_PRI_met", "log_DER_pt_ratio_lep_tau"]] = np.log(
    df[["PRI_met_sumet", "PRI_met", "DER_pt_ratio_lep_tau"]]
)

columns = ["log_PRI_met_sumet", "log_PRI_met", "log_DER_pt_ratio_lep_tau"]

df["Y"] = df["Label"].map({
    "s": 1,
    "b": 0,
})

from sklearn.model_selection import train_test_split

# Разобьем наши данные
X_train, X_test, y_train, y_test = train_test_split(
    df[columns + ["Weight"]].to_numpy(), df["Y"].to_numpy(),
    test_size=0.3, random_state=13)

# Отщепляем последний столбец

w_train = X_train[:, -1]
w_test = X_test[:, -1]

X_train = X_train[:, :-1]
```

5. Решающие деревья

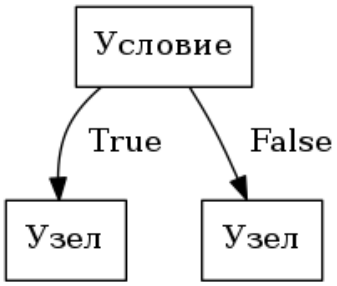
Решающее дерево

Решающее дерево (Decision Tree) - инструмент принятия решений. Широко используется в различных областях, в том числе и машинном обучении.

Стоит ли идти на улицу?



Бинарное дерево



Задачи

- **Классификация** - в листьях дерева окажется признак указывающий на класс, к которому относится объект
- **Регрессия** - в листьях окажется значение целевой функции в зависимости от признаков объекта.

Вопросы

- Как построить решающее дерево?
- Какие вопросы задавать дереву?
- Какие значения признаков выбрать?

Допустим нужно отгадать знаменитость, задавая вопросы. Ответ может быть либо **"Да"**, либо **"Нет"**.

Что лучше?

"Это Ричард Фейман?" **Vs** "Это женщина?"

Лучше тот, что дает нам наибольшую информацию.

Энтропия

Для системы с N состояниями можно определить энтропию Шеннона:

$$S = -\sum_{i=1}^{N} p_i \log_2 p_i$$

Как строить дерево?

Здесь (https://habr.com/ru/post/171759/) можно посмотреть пошаговый пример построения дерева.

Гиперпараметры

- max_depth - максимальная глубина дерева
- max_features - максимальное число признаков, по которым ищется лучшее разбиение
- min_samples_leaf - минимальное число объектов в листе (лист должен быть верен как минимум для этого числа прецендентов в обучающей выборке)

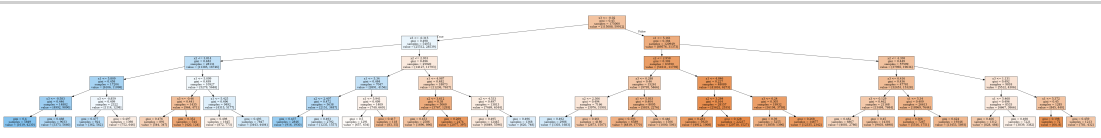
Пример

In [34]:
```
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import accuracy_score

tree = DecisionTreeClassifier(max_depth=5)
tree.fit(X_train, y_train)
predict = tree.predict(X_test)

print("Accuracy =", accuracy_score(y_test, predict))
```

```
Accuracy = 0.6991333333333334
AMS = 0.9304383952199664
```

Визуализация

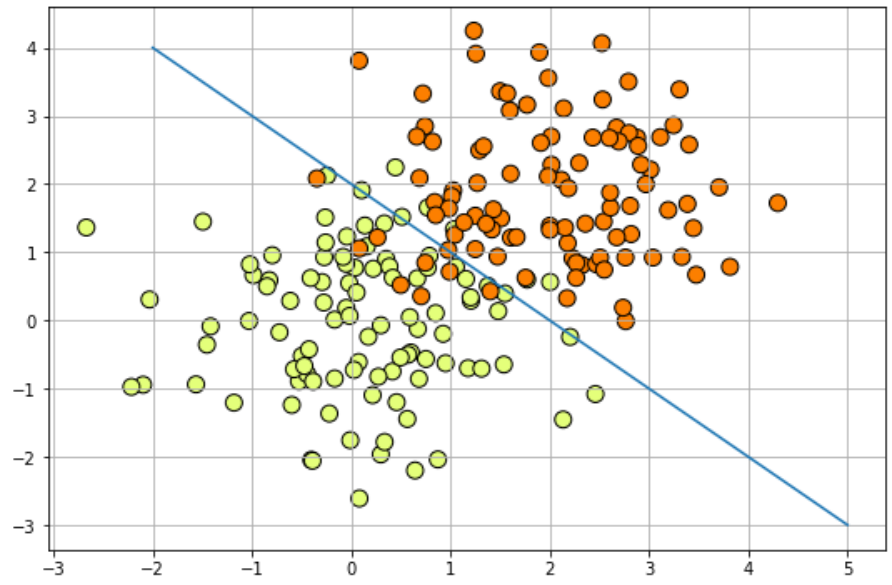
In [35]:
```
from sklearn.tree import export_graphviz
export_graphviz(tree, feature_names=['x1', 'x2', 'x3'],
                out_file="sktree.dot", filled=True)
```



Что на самом деле происходит?

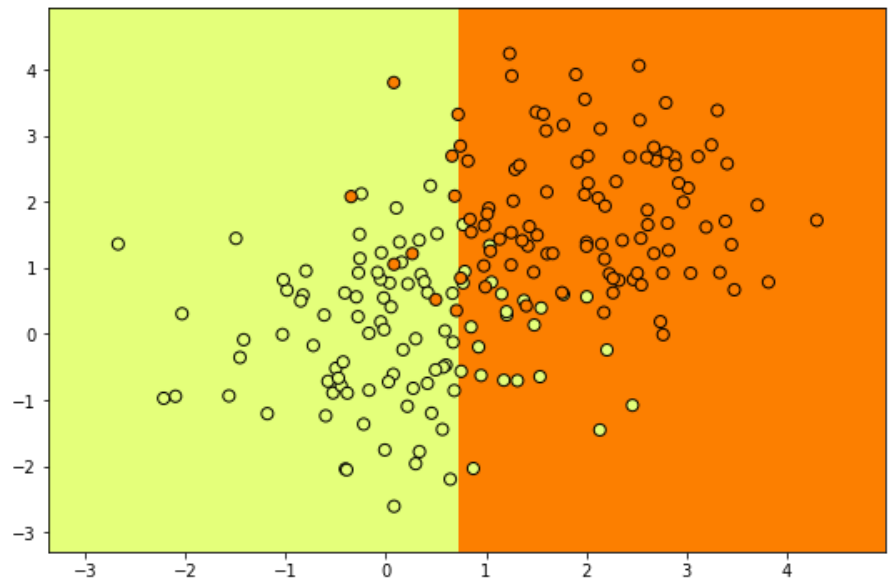
In [36]:
```
# Создадим два набора объектов, распределенных по Гауссу
train1 = np.random.normal(size=(100, 2))
train2 = np.random.normal(2, size=(100, 2))
data = np.vstack([train1, train2])
```

In [37]:
```
plt.scatter(data[:, 0], data[:, 1], c=target,
            cmap=plt.get_cmap("Wistia"),
            edgecolors='k', s=100, vmin=0, vmax=1);
plt.plot(np.linspace(-2, 5, 10), np.linspace(4, -3, 10))
```

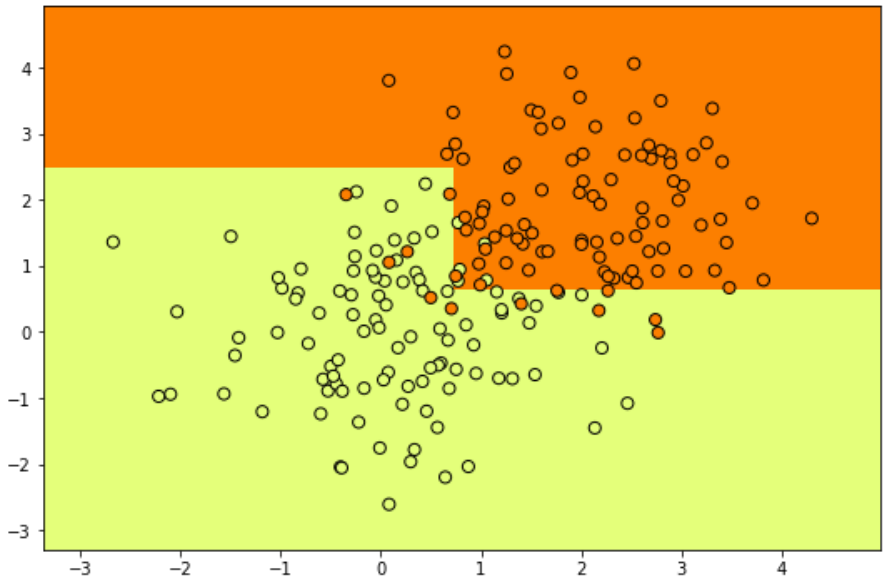


In [38]:
```
def tree_scatter(data, target, depth, outer=0.1):
    x_min, x_max = data[:, 0].min(), data[:, 0].max()
    x_w = x_max - x_min
    y_min, y_max = data[:, 1].min(), data[:, 1].max()
    y_w = y_max - y_min
    xx, yy = np.meshgrid(
        np.arange(x_min - x_w * outer, x_max + x_w * outer, 0.05),
        np.arange(y_min - y_w * outer, x_max + y_w * outer, 0.05),
    )

    tree = DecisionTreeClassifier(max_depth=depth)
    tree.fit(data, target)
    predict = tree.predict(np.stack([xx.ravel(), yy.ravel()], axis=1))
    plt.pcolormesh(xx, yy, predict.reshape(xx.shape), cmap=plt.get_cmap("Wistia
    plt.scatter(data[:, 0], data[:, 1], c=target,
                cmap=plt.get_cmap("Wistia"),
                edgecolors='k', s=50, vmin=0, vmax=1);
```

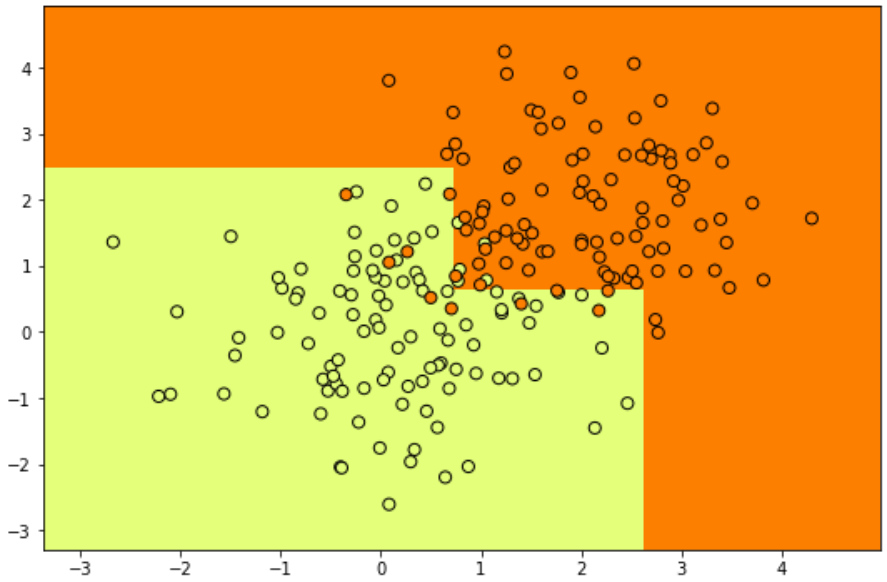
In [39]:
```
```



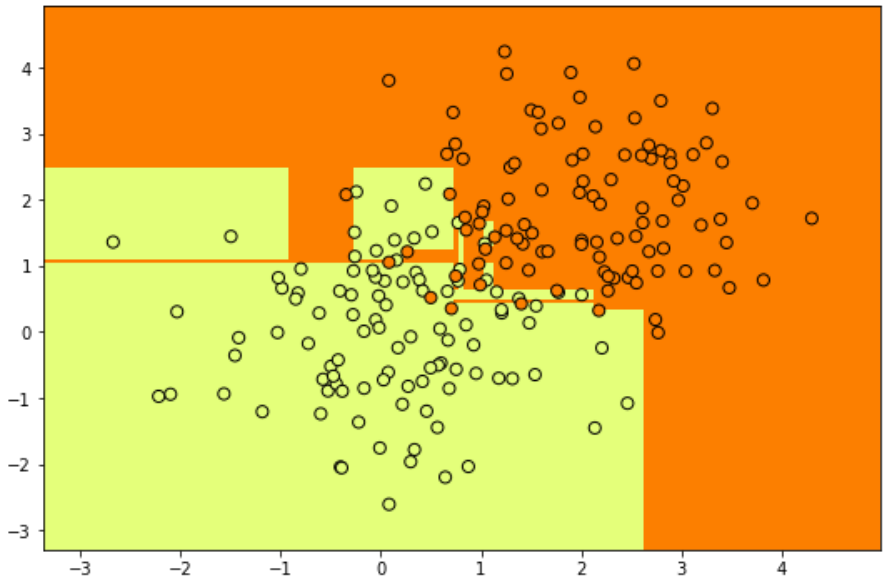
In [40]:



In [41]:

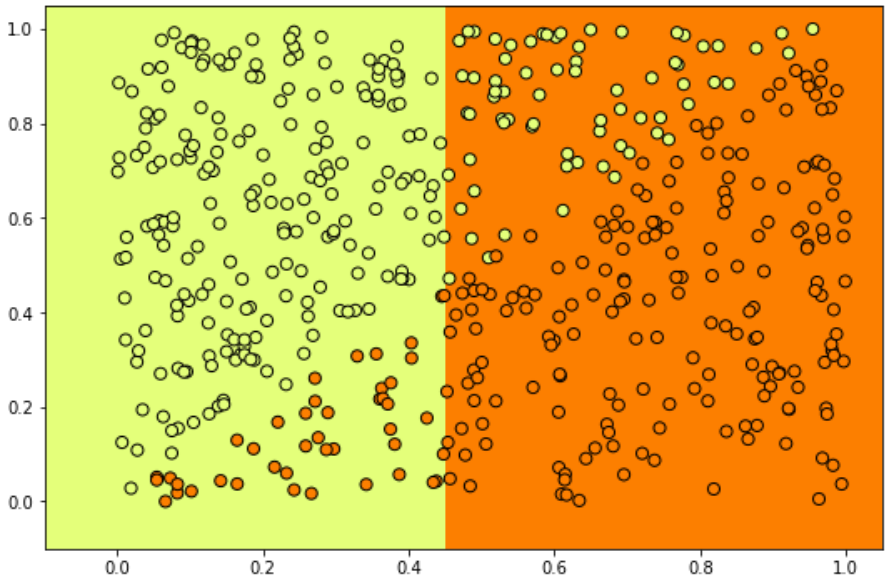


In [42]:

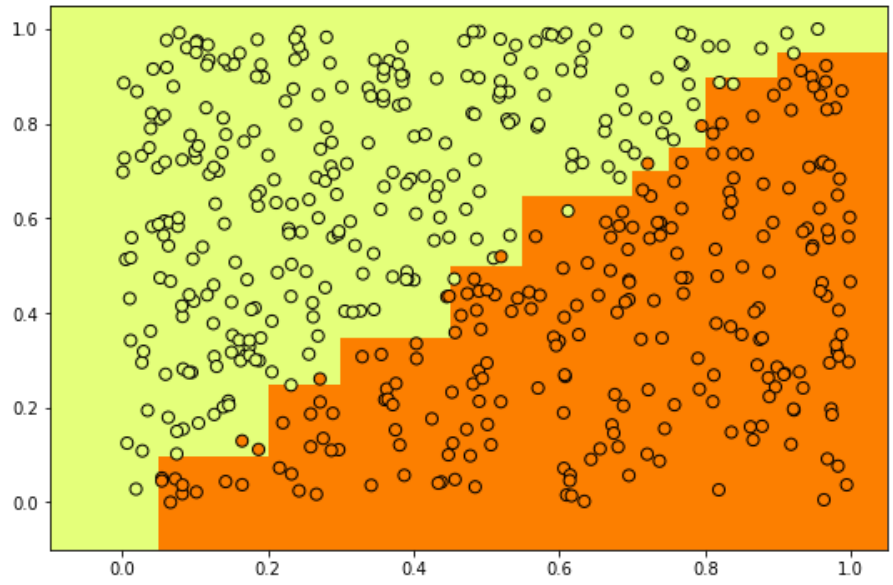


In [43]:
```
# А теперь рассмотрим сложный случай

data = np.random.uniform(size=(500, 2))
target = np.zeros(data.shape[0])
```

In [44]:



In [45]:



Основные выводы

Плюсы

- простота интерпретации (дерево - это фактически набор простых правил)
- высокая скорость обучения и прогнозирования
- небольшое число параметров модели
- не требует предварительной подготовки данных
- поддержка всех типов признаков

Минусы

- чувствительность к шумам или к изменениям данных (приводит к порой к кардинально другому дереву)
- разделяющая классы граница не всегда строится эффективно
- ограничение глубины дерева или отсечения ветвей (проблема переобучения)
- сложно поддерживать пропуски в данных
- плохо работает при большом числе признаков
- деревья умеют только интерполировать
- **данный алгоритм сам по себе почти никогда не применяется**

6. Ансамбли

Ансамбли

Ансамбль(фр. ensemble - совокупность, стройное целое) - это совокупность частей, образующих целое.

Мудрость Толпы (Wisdom of crowds)

Френсис Галтон посетил как-то рынок скота, на котором около 800 человек пытались отгадать

вес быка, стоявшего перед ними. Бык весил 1198 фунтов, но ни один человек не смог отгадать точный вес. При этом самое удивительное то, что если посчитать среднее от всех названных значений, то получится число очень близкое к реальному - 1197.

Голосование

- Простое голосование

$$R(f_1(\vec{x}), \dots, f_M(\vec{x})) = \frac{1}{M}\sum_{m=1}^{M} f_m(\vec{x})$$

- Взвешенное голосование

$$R(f_1(\vec{x}), \dots, f_M(\vec{x})) = \sum_{m=1}^{M} \omega_m f_m(\vec{x})$$

- Смесь

$$R(f_1(\vec{x}), \dots, f_M(\vec{x})) = \sum_{m=1}^{M} g_m(\vec{x}) f_m(\vec{x})$$

Хорошо, мы убедились, что ансамбли это хорошо, но у нас один набор данных, и, при этом, не всегда большой.

Также, у нас не так много методов, чтобы образовать многочисленное жюри.

Bootstrap

Класс методов, основанных на многократной генерации выборок методом Монте-Карло на основе уже имеющейся выборки.

По сути, мы строим эмпирическое распределение на основе представленных данных. Далее, мы его используем для генерации значений случайной величины, распределенных по построенному распределению.

Практически, мы просто случайным образом выбираем прецеденты из $\mathbf{X}$ с возвращением до тех пор, пока не получим выборку нужного размера (обычно того же, что и изначальная).

Бэггинг

Bagging (от Bootstrap aggregation) - один из самых простых способов построения ансамбля.

1. Пусть у нас есть набор прецедентов $\mathbf{X}$ размером N.
2. Мы случайно выбираем из него M элементов с возвращением, что в итоге дает нам новые подвыборки: $X_1, \dots, X_M$.
3. На каждом наборе $X_1, \dots, X_M$ мы обучаем наш алгоритм (базовые алгоритмы или слабые классификаторы), получая набор обученных моделей: $f_1(\vec{x}), \dots, f_M(\vec{x})$.
4. Итоговая классификация/регрессия - это простое голосование.

Почему?

- благодаря разнообразию обученых моделей, их ошибки взаимно компенсируются при голосовании
- выбросы могут не попадать в некоторые обучающие подвыборки
- итоговая модель обладает меньшей дисперсией, по сравнению с отдельными базовыми моделями

sklearn

В **sklearn** уже реализованы **BaggingClassifier** и **BagginRegressor**, которые могут использовать большинство других алгоритмов.

Метод случайных подпространст

Random subspace method(RSM) - в данном методе алгоритмы обучаются на различных подмножествах признаков, которые формируются случайным образом.

Может быть эффективен при большом числе признаков, небольшом числе объектов или наличие малоинформативных признаков.

Случайный лес

Дерево решений является очень хорошим семейством базовых классификаторов, для бэггинга.

1. Пусть K - число признаков, M - число выборок сгенерированных с помощью бутстрапа: $X_1, \dots, X_m$.
2. Построить дерево решений (обычно максимальной глубины) для каждого X_m, используя при каждом разбиении только k случайных признаков из K.
3. Для определение принадлежности прецедента к какому либо классу, проводится голосование среди M - базовых моделей.

Out-of-bag error

Пусть у нас есть выборка из N объектов. Следовательно вероятность достать из неё любой объект - это $\frac{1}{N}$. Нам нужно собрать новую подвыборку размером N доставая из ней объекты с возвращением.

Посчитаем вероятность того, что объект не попадет в подвыборку, что соответствует тому, что объект не взяли N раз.

$$p = \left(1 - \frac{1}{N}\right)^N$$

$$p = \lim_{N \to \infty} \left(1 - \frac{1}{N}\right)^N = \frac{1}{e}$$

из чего следует, что вероятность того, что конкретный объект попадет в подвыборку

$$p \approx 63\%$$

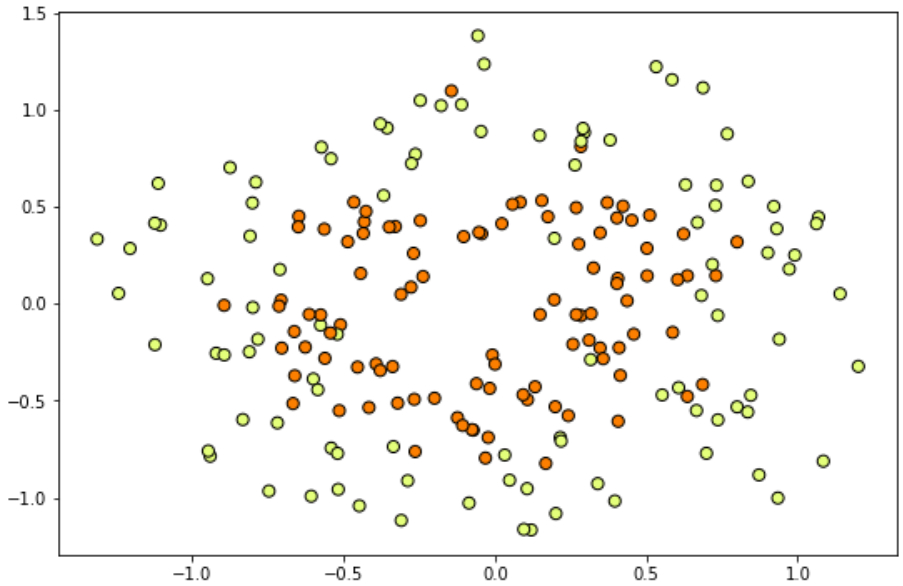
Отсюда следует, что в среднем 37% при бутстрапе остается в выборке, а значит мы можем использовать её для проверки нашей модели.

sklearn

В **sklearn** уже ализованы алгоритмы случайного леса **RandomForestClassifier** и **RandomForestRegressor**.

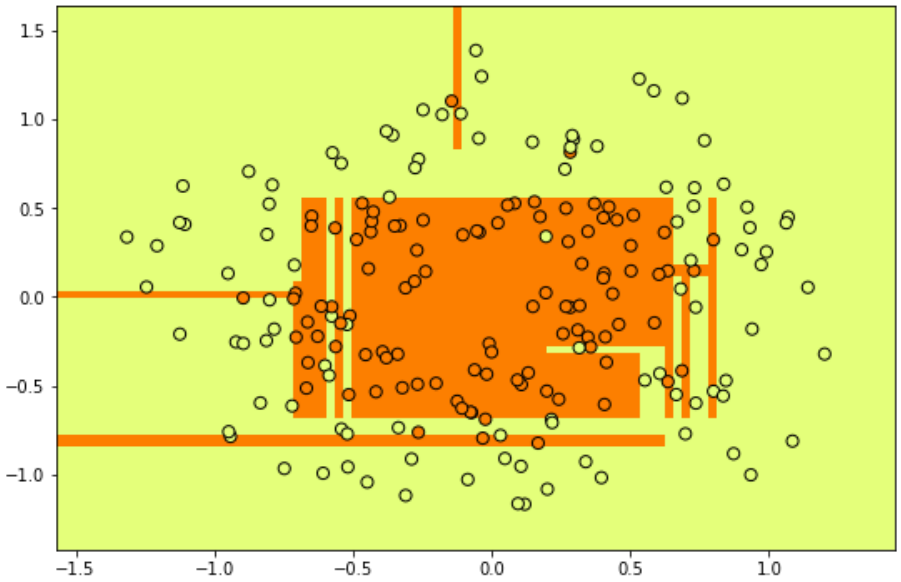
- n_estimators - число деревье
- max_depth - максимальная глубина дерева
- oob_score - рассчитывать ли out-of-bag error

In [19]:
```
from sklearn.datasets import make_circles, make_moons
np.random.seed(13)
#X, Y = make_circles(200, noise=0.2)
X, Y = make_circles(200, factor=0.5, noise=0.2)
plt.scatter(X[:, 0], X[:, 1], c=Y, cmap=plt.get_cmap("Wistia"),
            edgecolors='k', s=50)
```

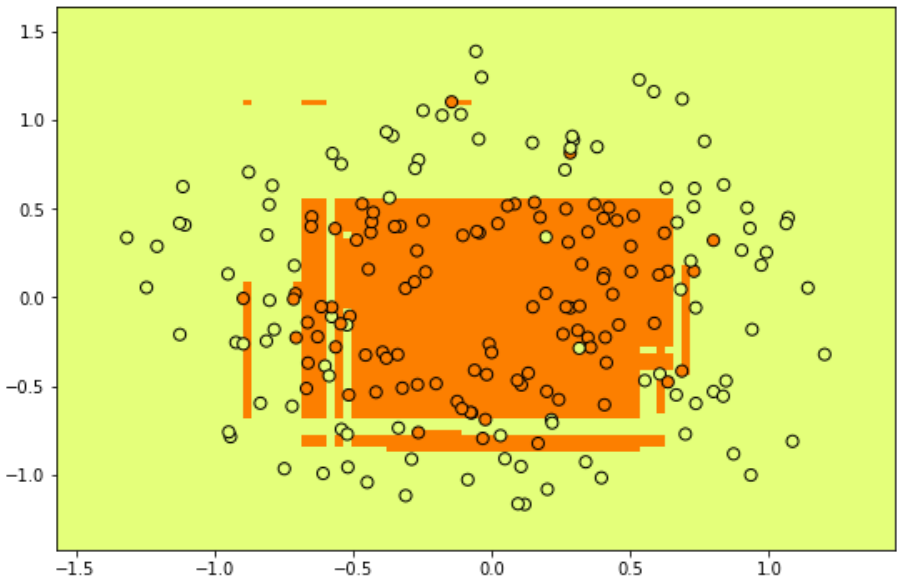


In [20]:
```
def plot(clf, X, Y):
    clf.fit(X, Y)

    x_min, x_max = X[:, 0].min(), X[:, 0].max()
    dx = (x_max - x_min) * 0.1
    y_min, y_max = X[:, 1].min(), X[:, 1].max()
    dy = (y_max - y_min) * 0.1
    xx, yy = np.meshgrid(np.linspace(x_min - dx, x_max + dx, 100),
                         np.linspace(y_min - dy, y_max + dy, 100))
    Z = clf.predict(np.stack([xx.ravel(), yy.ravel()], axis=1))
    Z = Z.reshape(xx.shape)

    plt.pcolormesh(xx, yy, Z, cmap=plt.cm.Wistia)
    plt.scatter(X[:, 0], X[:, 1], c=Y, cmap=plt.get_cmap("Wistia"),
            edgecolors='k', s=50)
```

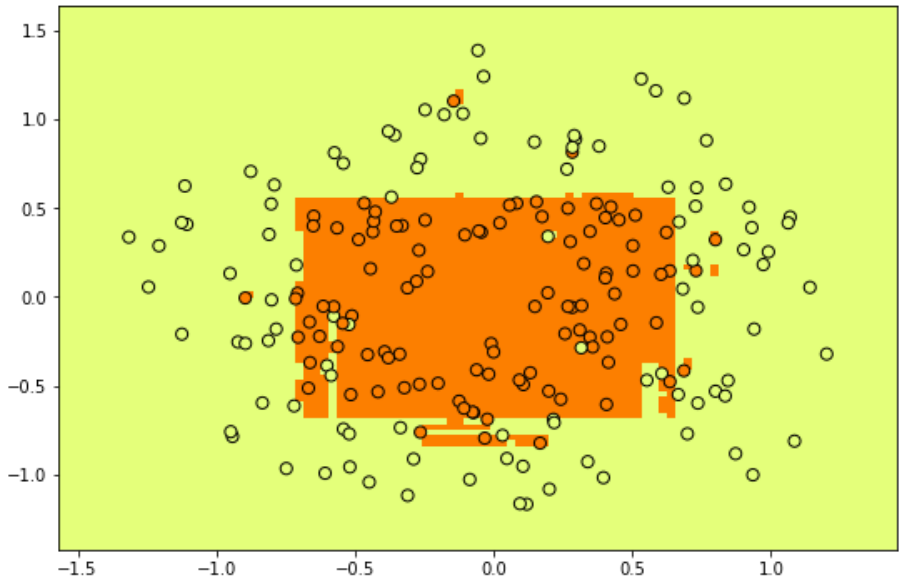
In [21]:
```
from sklearn.tree import DecisionTreeClassifier
```



In [22]:
```
from sklearn.ensemble import BaggingClassifier
```



In [23]:
```
from sklearn.ensemble import RandomForestClassifier
```



Пример

In [24]:
```
from sklearn.ensemble import RandomForestClassifier, BaggingClassifier

rfc = RandomForestClassifier(max_depth=5)
rfc.fit(X_train, y_train)
predict = rfc.predict(X_test)

print("Accuracy =", accuracy_score(y_test, predict))
```

```
Accuracy = 0.6997066666666667
AMS = 0.9420773287556513
```

In [25]:
```
# Приятная особенность леса: можно посмотреть важность признаков

feature_importances = pd.DataFrame(zip(columns, rfc.feature_importances_),
                                   columns=["Name", "Importance"])
```

Out[25]:

	Name	Importance
2	log_DER_pt_ratio_lep_tau	0.535948
0	log_PRI_met_sumet	0.254732
1	log_PRI_met	0.209320

In [26]:
```
dbc = BaggingClassifier(DecisionTreeClassifier(max_depth=5))
dbc.fit(X_train, y_train)
predict = dbc.predict(X_test)

print("Accuracy =", accuracy_score(y_test, predict))
```

```
Accuracy = 0.7003066666666666
AMS = 0.9387085131569476
```

Сверхслучайные деревья

Для тех, кому не достаточно случайности, можно использовать сверхслучайные деревья. В них порог для разбиения выбирается случайным образом. Позволяют уменьшить дисперсию, но ценой роста смещения.

В библиотеке sklearn они представлены классами **ExtraTreesClassifier** и **ExtraTreesRegressor**.

Плюсы случайного леса

- высокая точность предсказания, немного лишь уступающая бустингу
- не очень чувствителен к выбросам
- также, как и обычные деревья, не чувствителен к масштабу признаков
- весьма хорошо работает из коробки
- эффективно работает с большим числом признаков и классов
- сложно переобучить
- имеется возможность оценки важности признаков
- возможно распараллелить

Минусы случайного леса

- хуже работет в случае разреженных признаков
- также, как и обычные деревья, не умеет экстраполировать
- склонен к переобучению на зашумленных данных
- большой размер модели

Стэкинг (stacking)

Stacking - использует подход мета-классификации. Мы строим новую модель поверх множества моделей, и данная модель сама определяет наилучшую комбинацию базовых моделей.

- Пусть X - обучающая выборка, $f_1, \ldots, f_M$ - базовые классификаторы
- Формируем мета-классификатор, признаками для которого становятся метки классов, возвращаемые базовыми алгоритмами $f_1, \ldots, f_M$

Вариант 1 (Стекинг)

1. Делим обучающую выборку X на две подвыборки X_1 и X_2 (можно разбить и на большее количество фолдов).
2. Обучаем базовые модели на X_1 и делаем предсказания на X_2.
3. Обучаем те же базовые модели на X_2 и делаем предсказания на X_1.
4. Обучаем базовые модели на X и делаем предсказания на тестовой выборке.
5. Обучаем мета-модель на результатах (вероятностях) базовых моделей полученных в шаге 2 и 3.
6. Делаем предсказание мета-модели на данных с шага 4. Сравниваем результаты с истинными значениями тестовой выборки.

Иногда к признакам для мета-модели добавляют обычные признаки для базовых моделей.

Вариант 2 (Блендинг)

1. Делим обучающую выборку на основную X_m и на отложенную X_h(обычно порядка 10% от обучающей выборки).
2. Обучают несколько моделей на X_m. Делают предсказания P_h на отложенной выборке X_h и на тестовой выборке
3. Обучаем мета-модель на P_h и проверяем на предсказаниях, используя тестовую выборку

7. Метод k-ближайших соседей и метрики качества классификации

Метод k-ближайших соседей

Вторым из самых простых алгоритмов машинного обучения является метод k-ближайших соседей. Его основная идея заключается в том, что если какой-то объект близок по какому-нибудь критерию к другим объектам, то вероятно он также относится к категории этих объектов.

Как это работает?

Пусть у нас есть множество объектов $\mathbf{X}$ и набор целевых исходов/меток $\vec{Y}$.

Определим функцию растояния между объектами, такую, что:

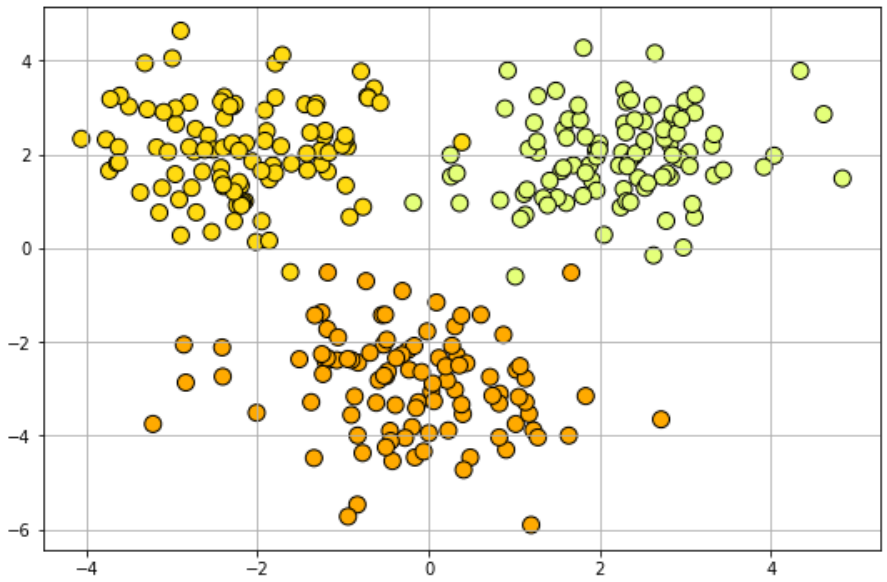
$$d(\vec{x}_1, \vec{x}_2) > 0, \text{ при } \vec{x}_1 \neq \vec{x}_2$$
$$d(\vec{x}_1, \vec{x}_2) = 0, \text{ при } \vec{x}_1 = \vec{x}_2$$
$$d(\vec{x}_1, \vec{x}_2) = d(\vec{x}_2, \vec{x}_1)$$
$$d(\vec{x}_1, \vec{x}_3) \leq d(\vec{x}_1, \vec{x}_2) + d(\vec{x}_2, \vec{x}_3)$$

Пусть у нас есть объект $\vec{x}$ и нам нужно определить, к какому классу относится данный объект.

Мы просто находим расстояние от $\vec{x}$ до всех прецедентов в обучающей выборке $\mathbf{X}$. Потом просто берем k объектов с наименьшим расстоянием до $\vec{x}$ и устраиваем голосование.

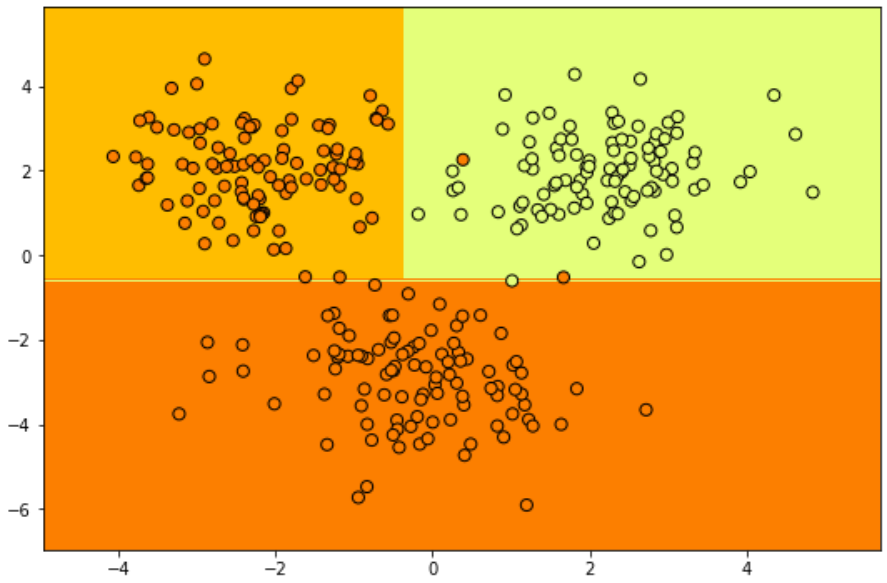
In [27]:
```
y1 = np.random.normal((2, 2), size=(100, 2))
y2 = np.random.normal((-2, 2), size=(100, 2))
y3 = np.random.normal((0, -3), size=(100, 2))
data = np.vstack([y1, y2, y3])
target = np.hstack([
    np.full(y1.shape[0], 0),
    np.full(y2.shape[0], 1),
    np.full(y3.shape[0], 2),
```

In [28]:
```
plt.scatter(data[:, 0], data[:, 1], c=target,
            cmap=plt.get_cmap("Wistia"), edgecolors='k', s=100,
            vmin=0, vmax=3)
plt.grid()
```

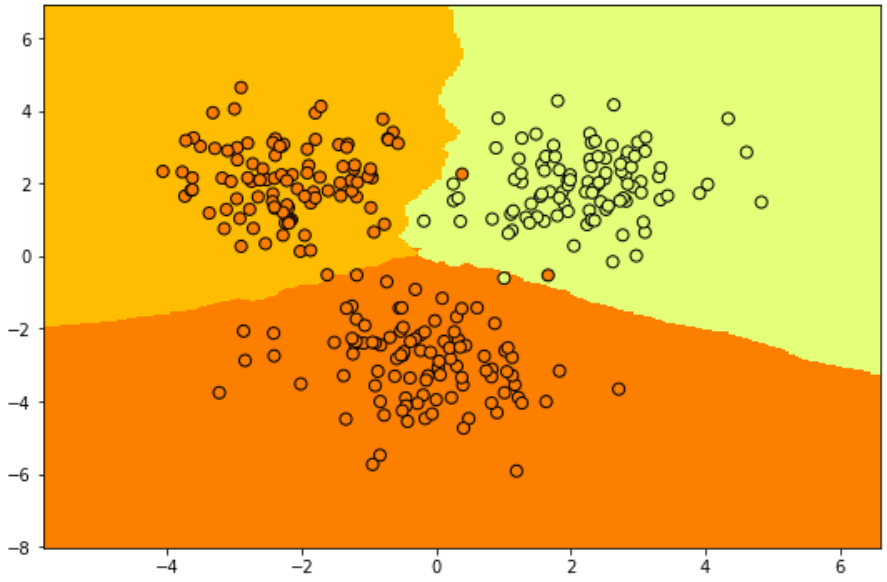


In [29]:
```
def plot_scatter(cls, data, target, outer=0.1, **kargs):
    x_min, x_max = data[:, 0].min(), data[:, 0].max()
    x_w = x_max - x_min
    y_min, y_max = data[:, 1].min(), data[:, 1].max()
    y_w = y_max - y_min
    xx, yy = np.meshgrid(
        np.arange(x_min - x_w * outer, x_max + x_w * outer, 0.05),
        np.arange(y_min - y_w * outer, x_max + y_w * outer, 0.05),
    )

    model = cls(**kargs)
    model.fit(data, target)
    predict = model.predict(np.stack([xx.ravel(), yy.ravel()], axis=1))
    plt.pcolormesh(xx, yy, predict.reshape(xx.shape), cmap=plt.get_cmap("Wistia
    plt.scatter(data[:, 0], data[:, 1], c=target,
                cmap=plt.get_cmap("Wistia"),
                edgecolors='k', s=50, vmin=0, vmax=1);
```

In [30]:
```
from sklearn.tree import DecisionTreeClassifier
plot_scatter(DecisionTreeClassifier, data, target,
```



In [31]:
```
from sklearn.neighbors import KNeighborsClassifier
plot_scatter(KNeighborsClassifier, data, target, 0.2,
```



Растояние

Как выбрать метрику расстояния между прецендентами?

Самое очевидно, но не всегда правильное, это использовать евклидову метрику

$$d(\vec{x}_1, \vec{x}_2) = \sqrt{\sum_i (x_{1i} - x_{2i})^2}$$

Есть множество различных метрик, наилучшую можно найти с помощью кросс-валидации

- Манхэттенское расстояние - $d(\vec{x}_1, \vec{x}_2) = \sum_i |x_{1i} - x_{2i}|$
- Расстояние Чебышева - $d(\vec{x}_1, \vec{x}_2) = \max_i |x_{1i} - x_{2i}|$
- Расстояние Минковского - $d(\vec{x}_1, \vec{x}_2) = \sqrt[1/p]{\sum_i (x_{1i} - x_{2i})^p}$
- Косинусное расстояние - $\frac{\vec{x}_1 \vec{x}_2}{\|\vec{x}_1\| \|\vec{x}_2\|}$

Подготовка данных

Что если один признак у нас принимает значения порядка 10^5, а второй признак порядка 10? Как это повлияет на определение расстояния?

Данные необходимо подготавливать

Минимакс

Один из самых простых подходов - это минимаксная нормализация

$$x^* = \frac{x - \min X}{\max X - \min X}$$

Это просто переводит значение признака на интервал $[0, 1]$

Стандартизация

Альтернативный подход - это масштабирование данных с помощью среднего значения и дисперсии

$$x^* = \frac{x - \bar{x}}{\sigma_x}$$

Голосование

Голосование за результирующий класс может быть

- невзвешенным (uniform) $v = \sum_n 1$
- взвешенным (distance) $v = \sum_n \frac{1}{d^2(\vec{x}, \vec{x}_n)}$

Растяжение осей

В случае наличия априорных сведения или банальной интуции, может стать известно, что один признак важнее другого признака. В этом случае мы можем ввести значимость признаков. Например, для евклидовой метрики этом примет вид

$$d(\vec{x}_1, \vec{x}_2) = \sqrt{\sum_i C_i (x_{1i} - x_{2i})^2}$$

Большие объемы данных

При больших объемах данных, вычисления уже начинают занимать существенное время.

- фильтрация обучающей выборки
- эффективная модель хранения данных

sklearn

В sklearn данным метод представлен двумя классами **KNeighborsClassifier** и **KNeighborsRegressor**.

Гиперпараметры

- *n_neighbors* - число соседей
- *metric* - метрика расстояния между прецендентами
- *weight* - веса

Плюсы

- хорошо изучен
- прост и понятен для интерпретации
- хороший старт для решения задач
- удобно использовать для построения мета-признаков или композиции алгоритмов
- применим в широком спектре задач

Минусы

- проклятие размерностей, при большом числе признаков, прецендеты в среднем оказываются рядом со всеми
- при больших выборках начинает медлено работать, можно исправить убиранием ненужных прецендентов, оптимальным представлением данных.
- сложность подбора метрики расстояния
- при небольшом числе соседей чувствителен к выбросам

Пример

In [48]:
```
from sklearn.neighbors import KNeighborsClassifier

knc = KNeighborsClassifier(n_neighbors=100)
knc.fit(X_train, y_train)
predict = knc.predict(X_test)

print("Accuracy =", accuracy_score(y_test, predict))
print("AMS =", AMS(w_test, y_test, predict))
```

```
Accuracy = 0.7023066666666666
AMS = 0.9516976634350779
```

In [49]:
```
# А теперь нормируем

from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import Pipeline

pipe = Pipeline([('scaler', StandardScaler()), ('knn', KNeighborsClassifier(n_n
pipe.fit(X_train, y_train)
predict = pipe.predict(X_test)

print("Accuracy =", accuracy_score(y_test, predict))
print("AMS =", AMS(w_test, y_test, predict))
```

```
Accuracy = 0.7023466666666667
AMS = 0.9516305770121326
```