MACHINE LEARNING ПРАКТИЧЕСКИЙ ПРИМЕР.

ПОДГОТОВКА ДАННЫХ

ПРИМЕНЕНИЕ АЛГОРИТМОВ ML

Машинное обучение — процесс, в результате которого машина (компьютер) способна показывать поведение, которое в нее не было *явно* заложено (запрограммировано).

A.L. Samuel Some Studies in Machine Learning Using the Game of Checkers // IBM Journal. July 1959. P. 210–229.

Загружаем данные

Задача - определение кредитной платежеспособности.

Пусть имеются данные о клиентах, обратившихся за кредитом.
Здесь:

- *объектами* являются клиенты
- *признаками* могут являться их пол, уровень дохода, образование, информация о том, имеется или отсутствует у них положительная кредитная история и т.д.

В качестве выделенного признака (*ответа*) выступает информация о том, вернул ли клиент кредит в банк или нет.

По этим данным требуется научиться предсказывать, вернет ли кредит новый клиент, обратившийся в банк.
Таким образом, речь идет о задаче **классификации**: требуется определить, какому классу: *положительному* (кредит будет возвращен) или *отрицательному* (кредит не будет возвращен) – принадлежит клиент.

В качестве модельных данных возьмем Credit Approval Data Set из коллекции UCI Machine Learning Repository.
Данные имеют формат csv и их можно скачать по указанному ниже адресу:

```
url = 'https://archive.ics.uci.edu/ml/machine-learning-databases/credit-screening/crx.data'
data = pd.read_csv(url, header=None, na_values='?')
```

Итак, мы загрузили данные в таблицу **data**. Объект data имеет тип **DataFrame** – это основной тип данных в библиотеке *pandas*, предназначенный для представления табличных данных

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	b	30.83	0.000	u	g	w	v	1.25	t	t	1	f	g	202	0	+
1	a	58.67	4.460	u	g	q	h	3.04	t	t	6	f	g	43	560	+
2	a	24.50	0.500	u	g	q	h	1.50	t	f	0	f	g	280	824	+
3	b	27.83	1.540	u	g	w	v	3.75	t	t	5	t	g	100	3	+
4	b	20.17	5.625	u	g	w	v	1.71	t	f	0	f	s	120	0	+

- строки соответствуют *объектам*
- столбцы – *признакам* этих объектов.

Объекты также называются *наблюдениями* или *примерами* (**samples**), а признаки – *атрибутами* (**features**).

Признаки бывают *количественными* (как, например, доход в рублях или рост в сантиметрах и т.д.) или *категориальными* (как, например, марка автомобиля, модель телефона и т.д.).

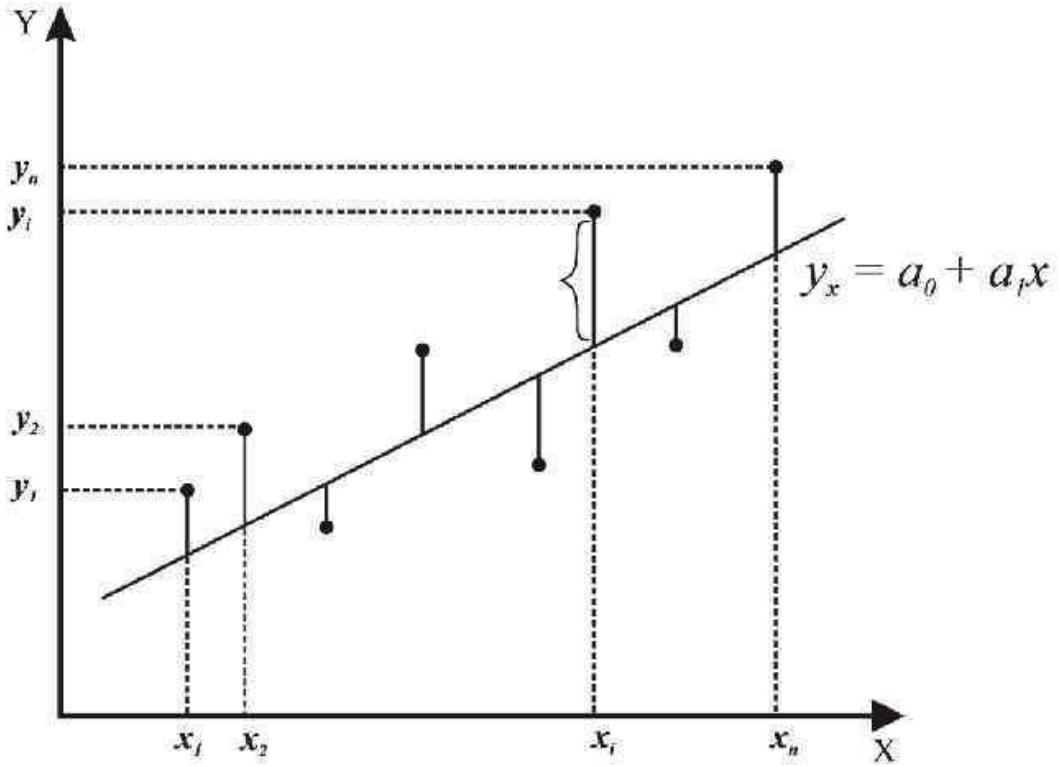
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	b	30.83	0.000	u	g	w	v	1.25	t	t	1	f	g	202	0	+
1	a	58.67	4.460	u	g	q	h	3.04	t	t	6	f	g	43	560	+
2	a	24.50	0.500	u	g	q	h	1.50	t	f	0	f	g	280	824	+
3	b	27.83	1.540	u	g	w	v	3.75	t	t	5	t	g	100	3	+
4	b	20.17	5.625	u	g	w	v	1.71	t	f	0	f	s	120	0	+

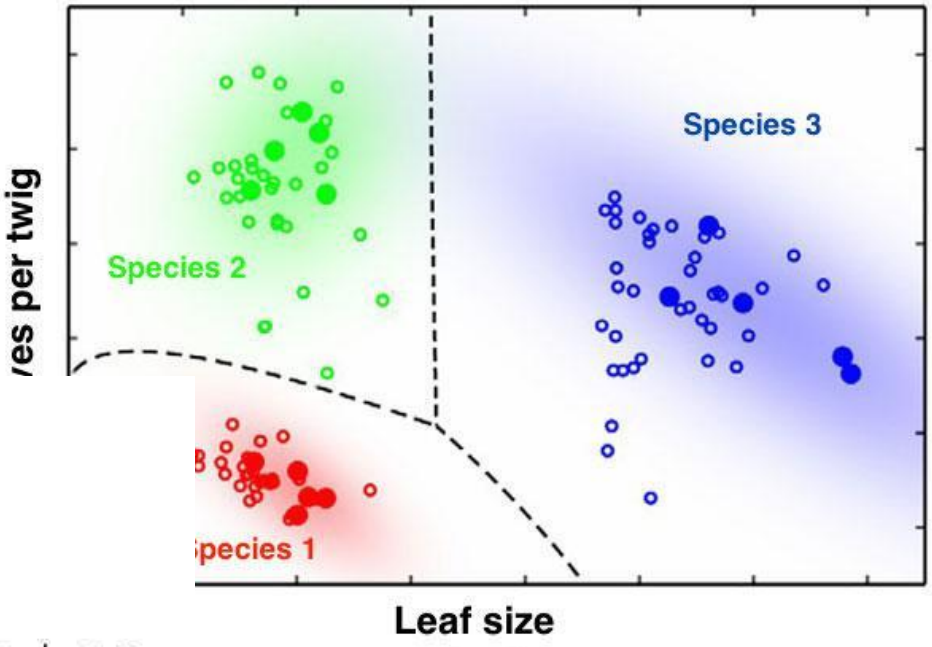
Входные признаки

Ответ

Требуется по имеющейся таблице *научиться* по новому объекту, которого нет в таблице, но для которого известны значения входных признаков, по возможности с небольшой ошибкой предсказывать значение выделенного признака (ответа).

- Если ответ количественный, то задача называется задачей *восстановления регрессии*.
- Если ответ категориальный, то задача называется задачей *классификации*.





Язык: python.
Библиотеки: numpy, pandas и scikit-learn.

- numpy содержит реализации многомерных массивов и алгоритмов линейной алгебры.
- pandas предоставляет широкий спектр функций по обработке табличных данных.
- scikit-learn реализует множество алгоритмов машинного обучения.

- matplotlib для научной визуализации.

Наиболее простой способ получить все требуемые библиотеки в python – установить дистрибутив anaconda.

Пример выполнен для anaconda 2.2.0 (с python версии 2.7).

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
plt.style.use('ggplot')
%matplotlib
```

Анализируем данные

Данные загружены. Попытаемся вначале их качественно проанализировать.
Узнаем размеры таблицы:

```
data.shape
```

```
(690, 16)
```

690 строк (*объектов*) и 16 столбцов (*признаков*), включая выходной (*целевой*) признак.
Мы можем посмотреть на несколько первых строк этой таблицы, чтобы получить представление об имеющихся данных:

```
data.head()
```

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	b	30.83	0.000	u	g	w	v	1.25	t	t	1	f	g	202	0	+
1	a	58.67	4.460	u	g	q	h	3.04	t	t	6	f	g	43	560	+
2	a	24.50	0.500	u	g	q	h	1.50	t	f	0	f	g	280	824	+
3	b	27.83	1.540	u	g	w	v	3.75	t	t	5	t	g	100	3	+
4	b	20.17	5.625	u	g	w	v	1.71	t	f	0	f	s	120	0	+

Строки таблицы пронумерованы числами от 0 до 689.
Столбцы не имеют каких-либо осмысленных имен и тоже просто пронумерованы.

Согласно описанию рассматриваемой задачи данные содержат информацию о клиентах, запрашивающих кредит. Для сохранения конфиденциальности данные обезличены, все значения категориальных признаков заменены символами, а числовые признаки приведены к другому масштабу. Последний столбец содержит символы + и -, соответствующие тому, вернул клиент кредит или нет.

Для удобства зададим столбцам имена:

```
data.columns = ['A' + (i) for i in (1, 16)] + ['class']
data.head()
```

	A1	A2	A3	A4	A5	A6	A7	A8	A9	A10	A11	A12	A13	A14	A15	class
0	b	30.83	0.000	u	g	w	v	1.25	t	t	1	f	g	202	0	+
1	a	58.67	4.460	u	g	q	h	3.04	t	t	6	f	g	43	560	+
2	a	24.50	0.500	u	g	q	h	1.50	t	f	0	f	g	280	824	+
3	b	27.83	1.540	u	g	w	v	3.75	t	t	5	t	g	100	3	+
4	b	20.17	5.625	u	g	w	v	1.71	t	f	0	f	s	120	0	+

data.describe()

	A2	A3	A8	A11	A14	A15
count	678.000000	690.000000	690.000000	690.00000	677.000000	690.000000
mean	31.568171	4.758725	2.223406	2.40000	184.014771	1017.385507
std	11.957862	4.978163	3.346513	4.86294	173.806768	5210.102598
min	13.750000	0.000000	0.000000	0.00000	0.000000	0.000000
25%	22.602500	1.000000	0.165000	0.00000	75.000000	0.000000
50%	28.460000	2.750000	1.000000	0.00000	160.000000	5.000000
75%	38.230000	7.207500	2.625000	3.00000	276.000000	395.500000
max	80.250000	28.000000	28.500000	67.00000	2000.000000	100000.000000

С помощью метода **describe()** получим некоторую сводную информацию по всей таблице. По умолчанию будет выдана информация только для количественных признаков.

Это:

- общее их количество (**count**)
- среднее значение (**mean**)
- стандартное отклонение (**std**)
- минимальное (**min**)
- максимальное (**max**) значения
- медиана (50%)
- значения нижнего (25%) и верхнего (75%) квартилей

Заметим, что количество элементов в столбцах **A2**, **A14** меньше общего количества объектов (690), что говорит о том, что эти столбцы содержат пропущенные значения.

Выделим числовые и категориальные признаки:

```
categorical_columns = [c for c in data.columns if data[c].dtype.name == 'object']
numerical_columns = [c for c in data.columns if data[c].dtype.name != 'object']
print categorical_columns
print numerical_columns
```

```
['A1', 'A4', 'A5', 'A6', 'A7', 'A9', 'A10', 'A12', 'A13', 'class'] ['A2', 'A3', 'A8', 'A11', 'A14', 'A15']
```

Теперь мы можем получить некоторую общую информацию по категориальным признакам:

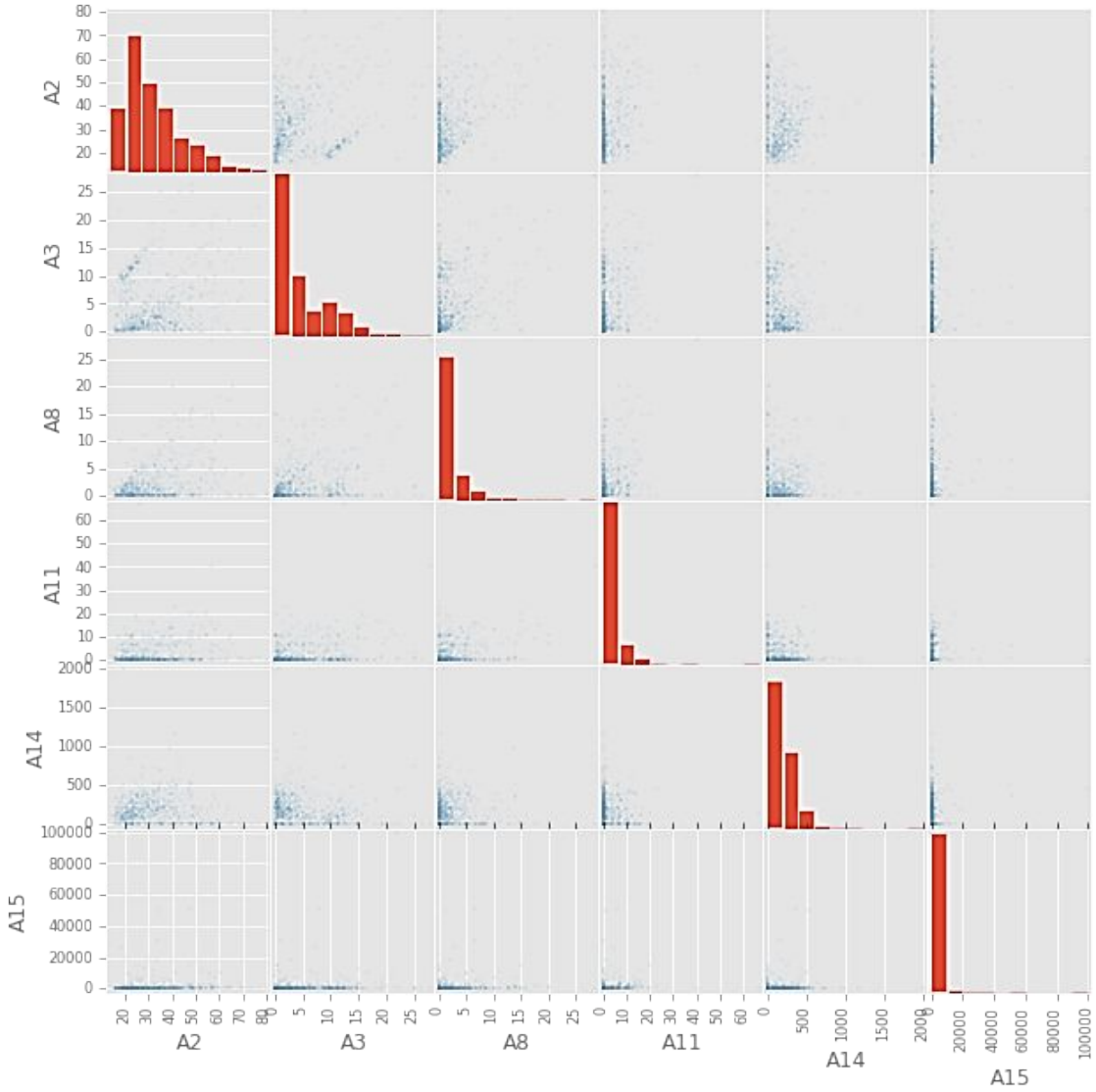
```
data[categorical_columns].describe()
```

	A1	A4	A5	A6	A7	A9	A10	A12	A13	class
count	678	684	684	681	681	690	690	690	690	690
unique	2	3	3	14	9	2	2	2	3	2
top	b	u	g	c	v	t	f	f	g	-
freq	468	519	519	137	399	361	395	374	625	383

В таблице для каждого категориального признака приведено общее число заполненных ячеек (**count**), количество значений, которые принимает данный признак (**unique**), самое популярное (часто встречающееся) значение этого признака (**top**) и количество объектов, в которых встречается самое частое значение данного признака (**freq**).

Функция **scatter_matrix** из модуля **pandas.tools.plotting** позволяет построить для каждой количественной переменной гистограмму, а для каждой пары таких переменных – диаграмму рассеяния:

```
from pandas.tools.plotting import scatter_matrix
scatter_matrix(data, alpha=0.05, figsize=(10, 10));
```



Из построенных диаграмм видно, что признаки не сильно коррелируют между собой

Готовим данные

Алгоритмы машинного обучения из библиотеки **scikit-learn** не работают напрямую с категориальными признаками и данными, в которых имеются пропущенные значения. Поэтому вначале подготовим наши данные.

Пропущенные значения

Узнать количество заполненных (непропущенных) элементов можно с помощью метода **count**. Параметр **axis = 0** указывает, что мы двигаемся по размерности 0 (сверху вниз), а не размерности 1 (слева направо), т.е. нас интересует количество заполненных элементов в каждом столбце, а не строке:

```
data.count(axis=0)
```

```
A1 678
A2 678
A3 690
A4 684
A5 684
A6 681
A7 681
A8 690
A9 690
A10 690
A11 690
A12 690
A13 690
A14 677
A15 690
class 690
dtype: int64
```

Если данные содержат пропущенные значения, то имеется две простые альтернативы (конкретно в нашем случае):

- удалить столбцы с такими значениями (**data = data.dropna(axis=1)**),
- удалить строки с такими значениями (**data = data.dropna(axis=0)**).

После этого, к сожалению, данных может стать совсем мало, поэтому рассмотрим простые альтернативные способы.

Также важно сохранение пропорциональности данных!

Количественные признаки

Заполним, например, медианными значениями:

```
data = data.fillna(data.median(axis=0), axis=0)
```

Категориальные признаки

Простая стратегия – заполнение пропущенных значений самым популярным в столбце:

```
data_describe = data.describe(include=[object])
for c in categorical_columns:
    data[c] = data[c].fillna(data_describe[c]['top'])
```

	A2	A3	A8	A11	A14	A15
count	690.000000	690.000000	690.000000	690.00000	690.000000	690.000000
mean	31.514116	4.758725	2.223406	2.40000	183.562319	1017.385507
std	11.860245	4.978163	3.346513	4.86294	172.190278	5210.102598
min	13.750000	0.000000	0.000000	0.00000	0.000000	0.000000
25%	22.670000	1.000000	0.165000	0.00000	80.000000	0.000000
50%	28.460000	2.750000	1.000000	0.00000	160.000000	5.000000
75%	37.707500	7.207500	2.625000	3.00000	272.000000	395.500000
max	80.250000	28.000000	28.500000	67.00000	2000.000000	100000.000000

Векторизация

Библиотека **scikit-learn** не умеет напрямую обрабатывать *категориальные* признаки.
Поэтому прежде чем подавать данные на вход алгоритмов машинного обучения преобразуем категориальные признаки в количественные.

Категориальные признаки, принимающие два значения (т.е. *бинарные* признаки) и принимающие большее количество значений будем обрабатывать по-разному.

Вначале выделим *бинарные* и *небинарные* признаки:

```
binary_columns = [c for c in categorical_columns if data_describe[c]['unique'] == 2]

nonbinary_columns = [c for c in categorical_columns if data_describe[c]['unique'] > 2]
```

Значения **бинарных** признаков просто заменим на 0 и 1

```
for c in binary_columns[1:]:
    top = data_describe[c]['top']
    top_items = data[c] == top
    data.loc[top_items, c] = 0
    data.loc[np.logical_not(top_items), c] = 1
```

К **небинарными** признакам применим метод *векторизации*, который заключается в следующем.
Признак **j**, принимающий **s** значений, заменим на **s** признаков, принимащих значения 0 или 1, в зависимости от того, чему равно значение исходного признака **j**.

```
data_nonbinary = pd.get_dummies(data[nonbinary_columns])

print data_nonbinary.columns
```

Нормализация количественных признаков

Многие алгоритмы машинного обучения чувствительны к масштабированию данных. К таким алгоритмам, например, относится метод ближайших соседей, машина опорных векторов и др. В этом случае количественные признаки полезно *нормализовать*. Это можно делать разными способами. Например, каждый количественный признак приведем к нулевому среднему и единичному среднеквадратичному отклонению:

```
data_numerical = data[numerical_columns]

data_numerical = (data_numerical - data_numerical.mean()) / data_numerical.std()

data_numerical.describe()
```

	A2	A3	A8	A11	A14	A15
count	6.900000e+02	6.900000e+02	6.900000e+02	6.900000e+02	6.900000e+02	690.000000
mean	-2.486900e-15	2.059544e-16	4.736952e-16	1.029772e-17	4.891417e-17	0.000000
std	1.000000e+00	1.000000e+00	1.000000e+00	1.000000e+00	1.000000e+00	1.000000
min	-1.497787e+00	-9.559198e-01	-6.643947e-01	-4.935286e-01	-1.066043e+00	-0.195272
25%	-7.456942e-01	-7.550425e-01	-6.150897e-01	-4.935286e-01	-6.014412e-01	-0.195272
50%	-2.575087e-01	-4.035072e-01	-3.655762e-01	-4.935286e-01	-1.368388e-01	-0.194312
75%	5.221970e-01	4.919034e-01	1.200038e-01	1.233822e-01	5.136044e-01	-0.119361
max	4.109180e+00	4.668645e+00	7.851932e+00	1.328414e+01	1.054901e+01	18.998208

И соединяем всё в одну таблицу.

```
data = pd.concat((data_numerical, data[binary_columns], data_nonbinary), axis=1)
data = pd.DataFrame(data, dtype=float)
```

Обучающая и тестовая выборки

Обучаться, или, как говорят, *строить модель*, мы будем на *обучающей выборке*, а проверять качество построенной модели – на *тестовой*.

Разбиение на тестовую и обучающую выборку должно быть случайным. Обычно используют разбиения в пропорции 50%:50%, 60%:40%, 75%:25% и т.д.
Мы воспользуемся функцией **train_test_split** из модуля **sklearn.cross_validation**. и разобьем данные на обучающую/тестовую выборки в отношении 70%:30%:

```
from sklearn.cross_validation import train_test_split

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.3, random_state = 11)

N_train, _ = X_train.shape

N_test, _ = X_test.shape
print N_train, N_test
```

Алгоритмы машинного обучения

Некоторые алгоритмы машинного обучения, реализованные в scikit-learn:

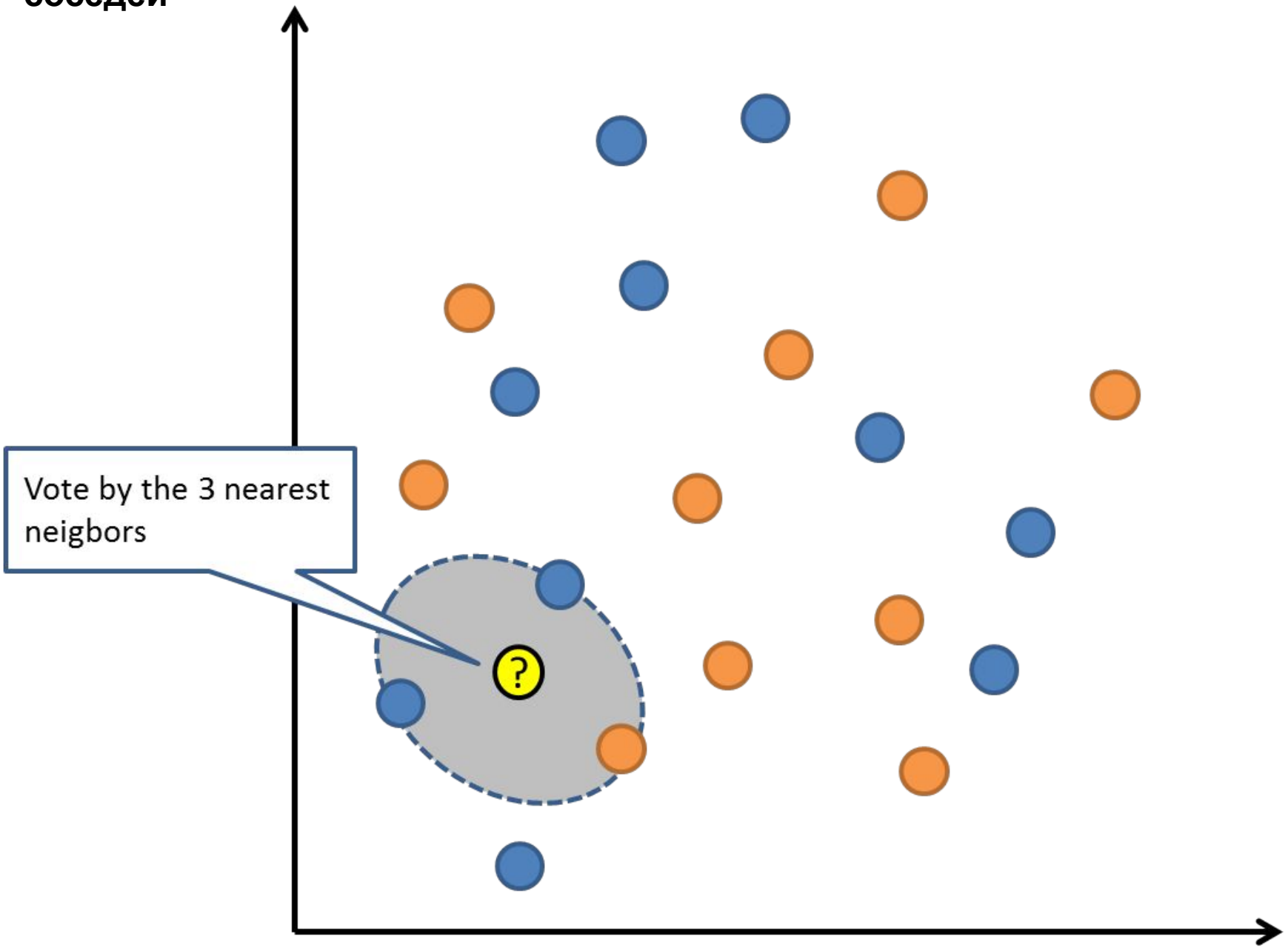
Метод	Класс
kNN – ***k*** ближайших соседей	**sklearn.neighbors.KNeighborsClassifier**
LDA – линейный дискриминантный анализ	**sklearn.lda.LDA**
QDA – квадратичный дискриминантный анализ	**sklearn.qda.QDA**
Logistic – логистическая регрессия	**sklearn.linear_model.LogisticRegression**
SVC – машина опорных векторов	**sklearn.svm.SVC**
Tree – деревья решений	**sklearn.tree.DecisionTreeClassifier**
RF – случайный лес	**sklearn.ensemble.RandomForestClassifier**
AdaBoost – адаптивный бустинг	**sklearn.ensemble.AdaBoostClassifier**
GBT – градиентный бустинг деревьев решений	**sklearn.ensemble.GradientBoostingClassifier**

Основные методы классов, реализующих алгоритмы машинного обучения

Все алгоритмы выполнены в виде классов, обладающих по крайней мере следующими методами:

Метод класса	Описание
fit(X, y)	обучение (тренировка) модели на обучающей выборке **X, y**
predict(X)	предсказание на данных **X**
set_params(params)**	установка параметров алгоритма
get_params()	чтение параметров алгоритма

*k*NN – метод ближайших соседей



*k*NN – метод ближайших соседей

Начнем с одного из самых простых алгоритмов машинного обучения – метода ***k** ближайших соседей* (***k*NN**).
Для нового объекта алгоритм ищет в обучающей выборке k наиболее близких объекта и относит новый объект к тому классу, которому принадлежит большинство из них.
Количество соседей ***k*** соответствует параметру **n_neighbors**. По умолчанию, **n_neighbors = 5**.
Вначале обучим модель:

```
from sklearn.neighbors import KNeighborsClassifier

knn = KNeighborsClassifier()

knn.fit(X_train, y_train)
```

После того, как модель обучена, мы можем предсказывать значение целевого признака по входным признакам для новых объектов. Делается это с помощью метода predict.
Нас интересует качество построенной модели, поэтому будем предсказывать значение выходного признака на тех данных, для которых оно известно: на обучающей и (что более важно) тестовой выборках:

```
y_train_predict = knn.predict(X_train)

y_test_predict = knn.predict(X_test)

err_train = np.mean(y_train != y_train_predict)

err_test = np.mean(y_test != y_test_predict)
print err_train, err_test
```

```
0.146997929607 0.169082125604
```

err_train и **err_test** – это ошибки на обучающей и тестовой выборках.
Как мы видим, они составили **14.7%** и **16.9%**.

Для нас более важной является ошибка на тестовой выборке, так как мы должны уметь предсказывать правильное (по возможности) значение на новых объектах, которые при обучении были недоступны.
Попробуем уменьшить тестовую ошибку, варьируя параметры метода.
Основной параметр метода ***k*** ближайших соседей – это ***k***.
Поиск оптимальных значений параметров можно осуществить с помощью класса **GridSearchCV** – поиск наилучшего набора параметров, доставляющих минимум ошибке перекрестного контроля (cross-validation).
По умолчанию рассматривается 3-кратный перекрестный контроль.
Например, найдем наилучшее значение ***k*** среди значений **[1, 3, 5, 7, 10, 15]**:

```
from sklearn.grid_search import GridSearchCV

n_neighbors_array = [1, 3, 5, 7, 10, 15]

knn = KNeighborsClassifier()

grid = GridSearchCV(knn, param_grid={'n_neighbors': n_neighbors_array})

grid.fit(X_train, y_train)

best_cv_err = 1 - grid.best_score_

best_n_neighbors = grid.best_estimator_.n_neighbors
print best_cv_err, best_n_neighbors
```

```
0.207039337474    7
```

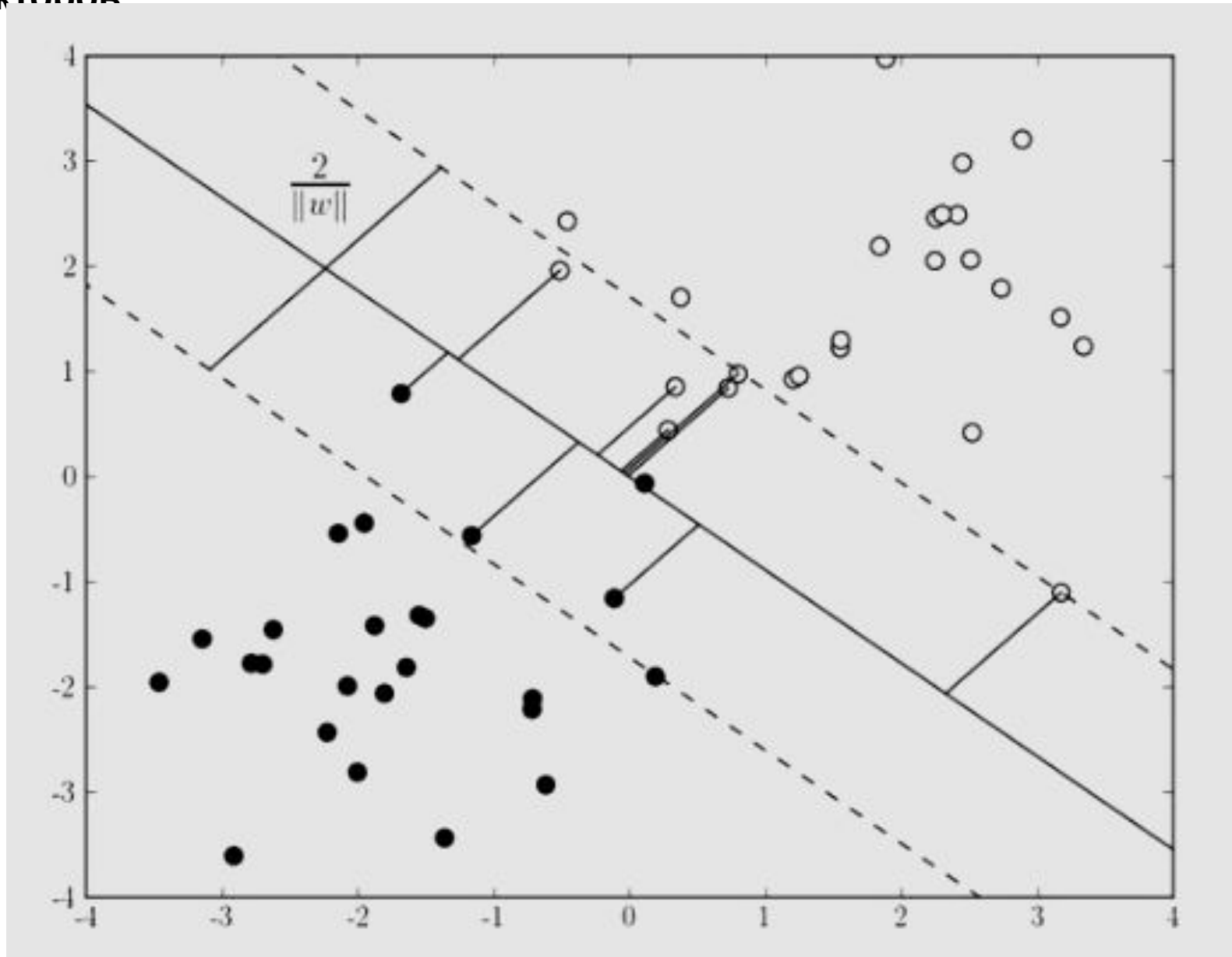
В качестве оптимального метод выбрал значение ***k*** равное 7. Ошибка перекрестного контроля составила 20.7%, что даже больше ошибки на тестовой выборке для 5 ближайших соседей. Это может быть обусленно тем, что для построения моделей в рамках схемы перекрестного контроля используются не все данные.
Проверим, чему равны ошибки на обучающей и тестовой выборках при этом значении параметра

```
knn = KNeighborsClassifier(n_neighbors=best_n_neighbors)
knn.fit(X_train, y_train)
err_train = np.mean(y_train != knn.predict(X_train))
err_test = np.mean(y_test != knn.predict(X_test)) print err_train, err_test
```

```
0.151138716356    0.164251207729
```

Как видим, метод ближайших соседей на этой задаче дает не слишком удовлетворительные результаты.

SVC – машина опорных векторов



SVC – машина опорных векторов

Следующий метод, который мы попробуем – машина опорных векторов (SVM – support vector machine или SVC – support vector classifier) – сразу приводит к более оптимистичным результатам.
Уже со значением параметров по умолчанию (в частности, ядро – радиальное **rbf**) получаем более низкую ошибку на обучающей выборке:

```
from sklearn.svm import SVC svc = SVC()

svc.fit(X_train, y_train)

err_train = np.mean(y_train != svc.predict(X_train))

err_test = np.mean(y_test != svc.predict(X_test))

print err_train, err_test
```

```
0.144927536232    0.130434782609
```

Итак, на тестовой выборке получили ошибку в **13%**.
С помощью подбора параметров попробуем ее еще уменьшить.

Рассмотрим подбор параметров только для радиального ядра в данном примере.

Радиальное

```
from sklearn.grid_search import GridSearchCV

C_array = np.logspace(-3, 3, num=7)

gamma_array = np.logspace(-5, 2, num=8)

svc = SVC(kernel='rbf')

grid = GridSearchCV(svc, param_grid={'C': C_array, 'gamma': gamma_array})

grid.fit(X_train, y_train) print 'CV error = ', 1 - grid.best_score_

print 'best C = ', grid.best_estimator_.C

print 'best gamma = ', grid.best_estimator_.gamma
```

```
CV error = 0.138716356108
best C = 1.0
best gamma = 0.01
```

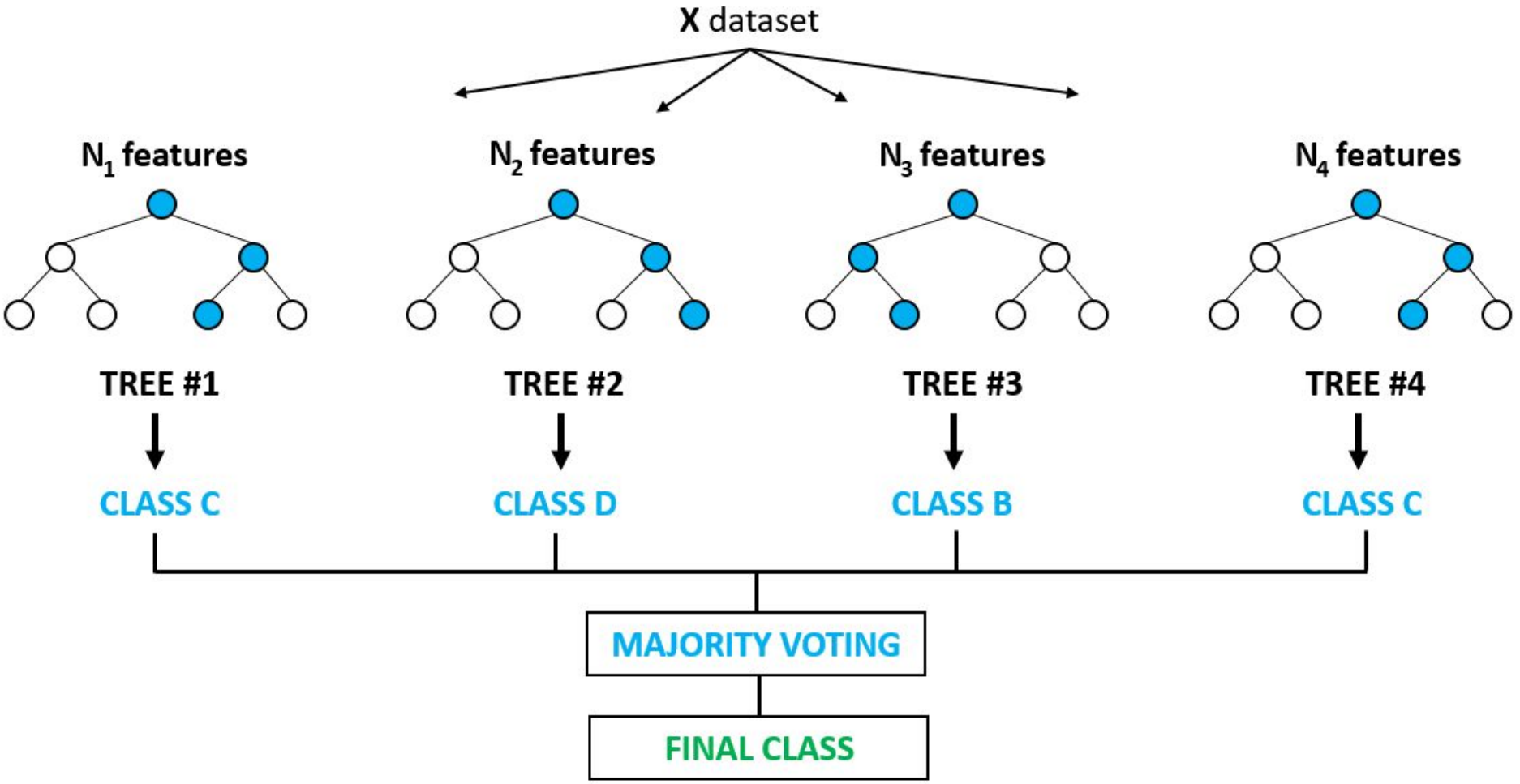
Получили ошибку перекрестного контроля в 13.9%.
Посмотрим, чему равна ошибка на тестовой выборке при найденных значениях параметров алгоритма:

```
svc = SVC(kernel='rbf', C=grid.best_estimator_.C, gamma=grid.best_estimator_.gamma)

svc.fit(X_train, y_train)

err_train = np.mean(y_train != svc.predict(X_train))

err_test = np.mean(y_test != svc.predict(X_test))

print err_train, err_test
```

```
0.134575569358 0.111111111111
```

Ошибка на тестовой выборке равна **11.1%**.
Заметно лучше, чем ***k*NN**!

Random Forest – случайный лес



Random Forest – случайный лес

Воспользуемся одним из самых популярных алгоритмов машинного обучения – случайный лес – Random Forest.

Алгоритм строит ансамбль случайных деревьев, каждое из которых обучается на выборке, полученной из исходной с помощью процедуры изъятия с возвращением.

```
from sklearn import ensemble

rf = ensemble.RandomForestClassifier(n_estimators=100, random_state=11)

rf.fit(X_train, y_train)

err_train = np.mean(y_train != rf.predict(X_train))

err_test = np.mean(y_test != rf.predict(X_test))

print err_train, err_test
```

```
0.0    0.0966183574879
```

Итак, ошибка на тестовой выборке составила **9.7%**.

Отбор признаков (Feature Selection) с помощью алгоритма случайного леса

Одной из важных процедур предобработки данных в алгоритмах их анализа является *отбор значимых признаков*. Его цель заключается в том, чтобы отобрать наиболее существенные признаки для решения рассматриваемой задачи классификации.

- Отбор признаков необходим для следующих целей:
- Для лучшего понимания задачи. Человеку легче разобраться с небольшим количеством признаков, чем с огромным их количеством.
- Для ускорения алгоритмов.
- Для улучшения качества предсказания. Устранение шумовых признаков может уменьшить ошибку алгоритма на тестовой выборке, т.е. улучшить качество предсказания.

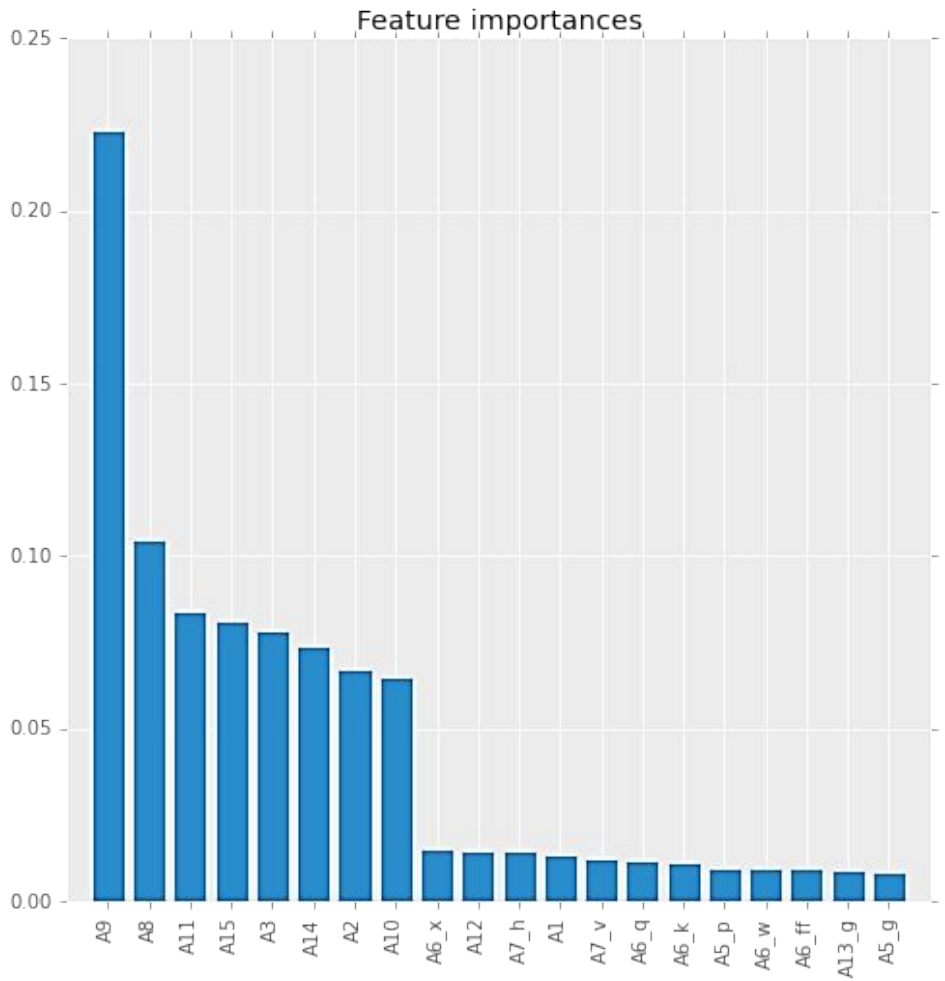
Отбор значимых признаков осуществляется как «*вручную*» — на основе анализа содержательной постановки задачи, так и «*автоматически*» — с помощью универсальных алгоритмов.

Отбор признаков «вручную» (как и «ручной» *синтез новых признаков*) — важный этап в анализе данных. К сожалению, нам не известны содержательные значения используемых в рассматриваемой задаче признаков, поэтому ограничимся только их автоматическим отбором.

Для этого существует много различных алгоритмов. Рассмотрим только один из них – с помощью случайного леса.
Все, что нужно сделать, – это после вызова метода **predict** для случайного леса прочитать поле **feature_importances_**. Для каждого признака это поле содержит число, выражающее «*важность*» этого признака. Чем больше число, тем значимее признак. Сумма всех чисел равна 1.
Упорядочим значимости и выведем их значения:

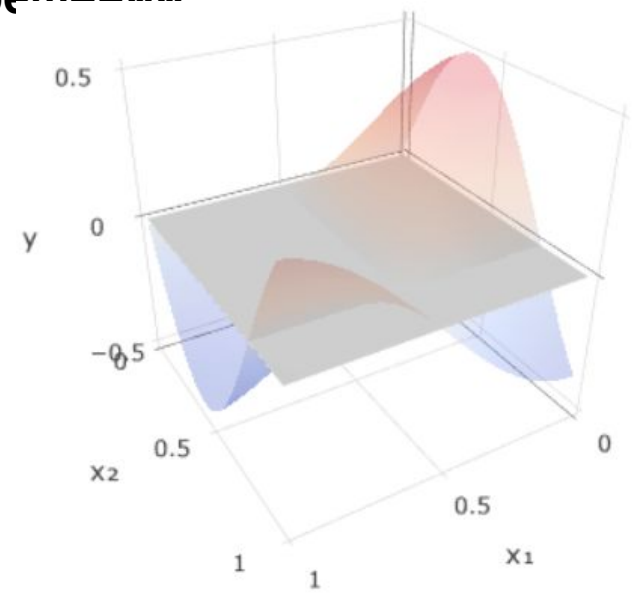
```
importances = rf.feature_importances_

indices = np.argsort(importances)[::-1]
print("Feature importances:")
for f, idx in enumerate(indices):
        print("{:2d}. feature '{:5s}' ({:.4f})".format(f + 1, feature_names[idx], importances[idx]))
```

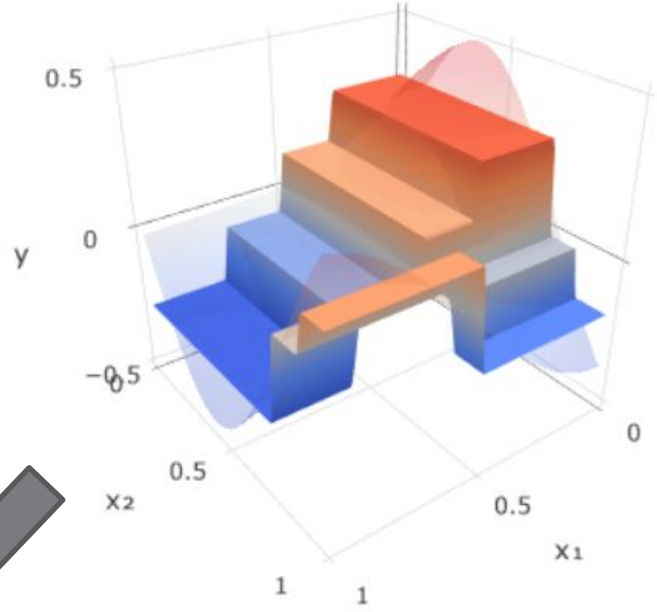
Построим столбцовую диаграмму, графически представляющую значимость первых 20 признаков:

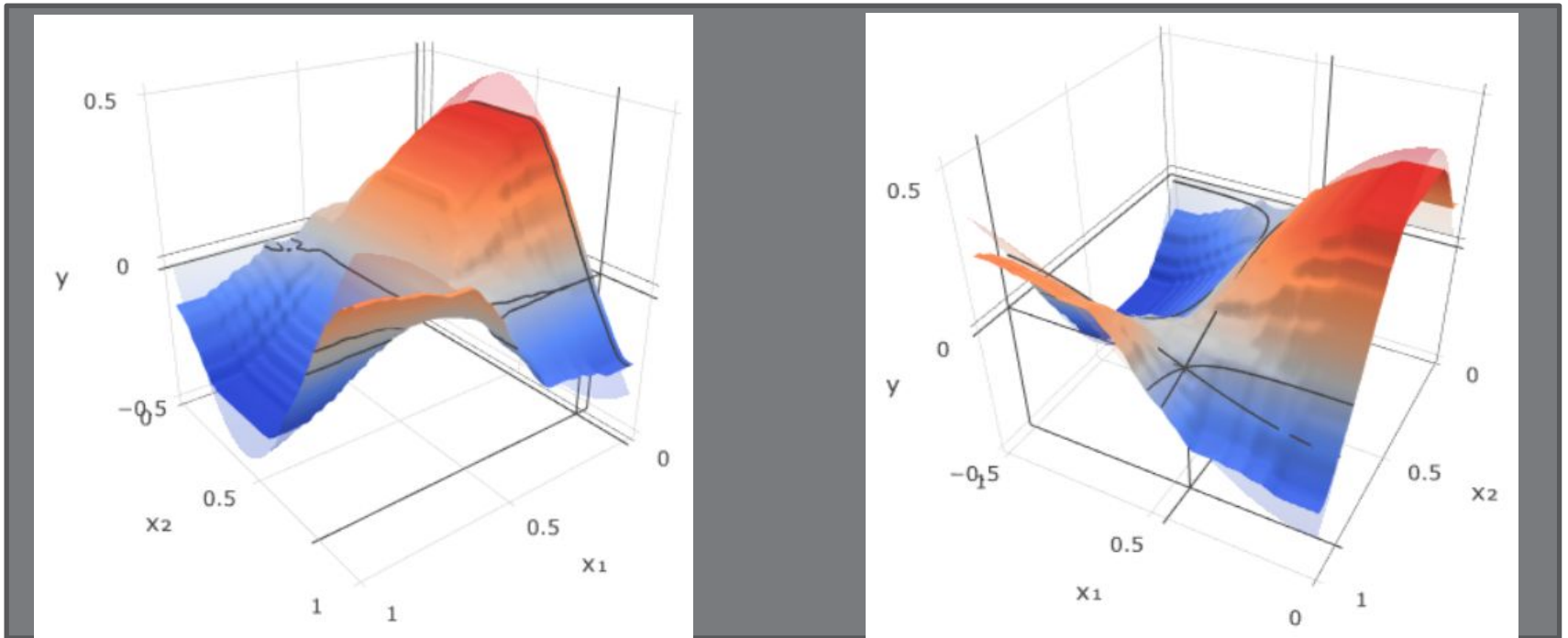


Мы видим, что основную роль играют признаки **A9, A8, A11, A15, A3, A14, A2, A10**. Теперь можно попытаться использовать только эти признаки для обучения других моделей.

GBT – градиентный бустинг деревьев решений







GBT – градиентный бустинг деревьев решений

GBT – еще один метод, строящий ансамбль деревьев решений. На каждой итерации строится новый классификатор, аппроксимирующий значение градиента функции потерь.

```
from sklearn import ensemble

gbt = ensemble.GradientBoostingClassifier(n_estimators=100, random_state=11)

gbt.fit(X_train, y_train)

err_train = np.mean(y_train != gbt.predict(X_train))

err_test = np.mean(y_test != gbt.predict(X_test))
print err_train, err_test
```

```
0.0248447204969  0.101449275362
```

Ошибка на тестовой выборке (**10.1%**) чуть выше, чем для случайного леса.
Заметим, что при использовании найденных выше значимых признаков ошибка практически не меняется:

```
0.0351966873706  0.106280193237
```

Итоги

- Победил алгоритм случайного леса – **Random Forest**, с тестовой ошибкой **9.7%**.
- Также неплохие результаты удалось получить с помощью градиентного бустинга деревьев решений – **GBT**. Причем кажется, что из GBT выжато не все, на что он способен.
- Можно продолжать эксперименты с этими и другими алгоритмами.

- Как уже отмечалось ранее, также важен отбор и синтез признаков. Для этого также можно попробовать разные алгоритмы.

- Ну и, разумеется, оценить работу модели необходимо на реальных данных.