In [1]:

```
from sklearn import *
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import pandas as pd
import sklearn as sk
```

In [2]:

```
X, y = sk.datasets.load_svmlight_file('a7a')
print(X.shape, y.shape)
X = np.asarray(X.todense())[:16000]
y = y[:16000]
print(y[:10])
print(X.shape, y.shape)
```

```
(16100, 122) (16100,)
[-1. -1. -1. -1.  1.  1. -1. -1. -1. -1.]
(16000, 122) (16000,)
```

In [28]:

```
class LogisticRegressor:
    def __init__(self, feature_num, eps=1e-2 / 2):
        self.eps = eps
        self.feature_num = feature_num
        self._reboot()

    def _reboot(self):
        self.weight = np.ones(self.feature_num)
        self.prev_weight = np.zeros(self.feature_num)
        self.iteration = 1
        self.alpha = None
        self.moment = None

    def fit(self, X, y, conditionality_coef, method, alpha_metod=None):
        if method != 'nesterov' and not alpha_metod:
            assert True, 'alpha method not provided'
        self._reboot()
        self._set_conditionality_coef(conditionality_coef, X)
        n = len(X)

        f_history = []
        grad_history = []
        w = []

        while not self._converged(method, X, y):
            f_history.append(self.loss(X, y, self.weight))
            grad_history.append((self.loss_gradient(X, y, method, self.weight)**2).s
            w.append(self.iteration)

            self.prev_weight = self.weight
            self.weight = self.step(X, y, method, alpha_metod)
            if method != 'stochastic':
                self.iteration += 1
            else:
                self.iteration += 1. / n

        return w, f_history, grad_history, self.weight

    def step(self, X, y, method, alpha_method):
        if method == 'gradient':
            alpha = self._get_alpha(alpha_method, X, y)
            h = self.loss_gradient(X, y, method, self.weight)
            return self.weight - alpha * h

        elif method == 'stochastic':
            alpha = self._get_alpha(alpha_method, X, y)
            h = self.loss_gradient(X, y, method, self.weight)
            return self.weight - alpha * h

        elif method == 'nesterov':
            self.moment = self.weight + ((self.L**0.5 - self.mu**0.5) / (self.L**0.5
            alpha = self._get_alpha(alpha_method, X, y)
            h = self.loss_gradient(X, y, method, self.weight)
            return self.moment - alpha * h

        else:
            raise NotImplementedError

    def loss(self, X, y, weight):
```

```
        value = -np.matmul(X, weight) * y
        loss = np.log(1 + np.exp(value)).mean()
        loss += (weight**2).sum() * self.mu / 2
        return loss

    def loss_gradient(self, X, y, method, t):
        if method != 'stochastic':
            grad = self.mu * t
            Ax = np.matmul(X, t)
            mat = (np.exp(-y * Ax) * (-y) / (1 + np.exp(-y * Ax))) / len(X)
            grad += np.matmul(mat, X)
            return grad
        else:
            index = np.random.randint(1, len(X))
            grad = self.mu * t
            Ax = np.matmul(X, t)
            mat = (np.exp(-y * Ax) * (-y) / (1 + np.exp(-y * Ax)))
            grad += mat[index] * X[index, :]
            return grad

    def get_objective_L(self, X):
        A = np.square(X)
        A = A.sum(axis=1)
        return np.max(A) / 4

    def _converged(self, method, X, y):
        if method == 'srochastic':
            return abs(((self.weight - self.prev_weight)**2).sum())**0.5 < self.eps
        else:
            return abs(self.loss(X, y, self.weight) - self.loss(X, y, self.prev_weig

    def _set_conditionality_coef(self, conditionality_coef, X):
        objective_L = self.get_objective_L(X)
        self.mu = objective_L / (1 / conditionality_coef - 1)
        self.L = self.mu + objective_L

    def _get_alpha(self, alpha_metod, X, y):
        if self.alpha is None:
            self.alpha = 1 / self.L

        if alpha_metod == 'const':
            return self.alpha
        elif alpha_metod == 'armiho':
            eps = 0.5
            theta = 0.9
            h = self.loss_gradient(X, y, 'gradient', self.weight)

            while self.loss(X, y, self.weight - h * self.alpha) - self.loss(X, y, se
                self.alpha *= theta
            return self.alpha
        elif alpha_metod == 'apriori':
            return (10 * self.alpha) / (self.iteration**0.5)
        else:
            raise NotImplementedError
```

In [30]:

```
reg = LogisticRegressor(122)
```

In [31]:

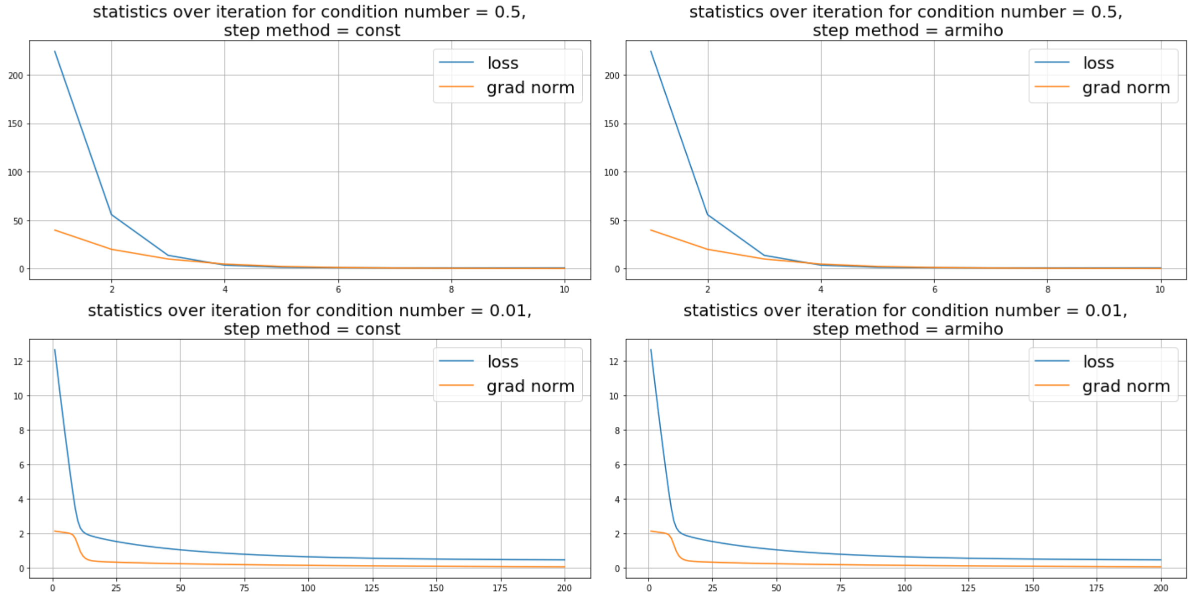
```
def plot(xs, ys, title, labels=None):
    plt.title(title, fontsize=20)
    for i, x, y in zip(range(len(xs)), xs, ys):
        if labels:
            plt.plot(x, y, label=labels[i])
        else:
            plt.plot(x, y)
    plt.grid(True)
    plt.legend(fontsize=20)
```

Gradient descent, different step methods and condition number comparison

In [37]:

```
plt.figure(figsize=(20, 10))
for i, conditionality_coef in enumerate([0.5, 0.01]):
    for j, alpha_method in enumerate(['const', 'armiho']):
        w, f_history, grad_history, weight = reg.fit(X, y, conditionality_coef=cond:
        plt.subplot(2, 2, 2 * i + j + 1)
        plot([w, w], [f_history, grad_history],
         'statistics over iteration for condition number = {},\n step method = {}'.f
         labels=['loss', 'grad norm'])

plt.tight_layout()
plt.show()
```

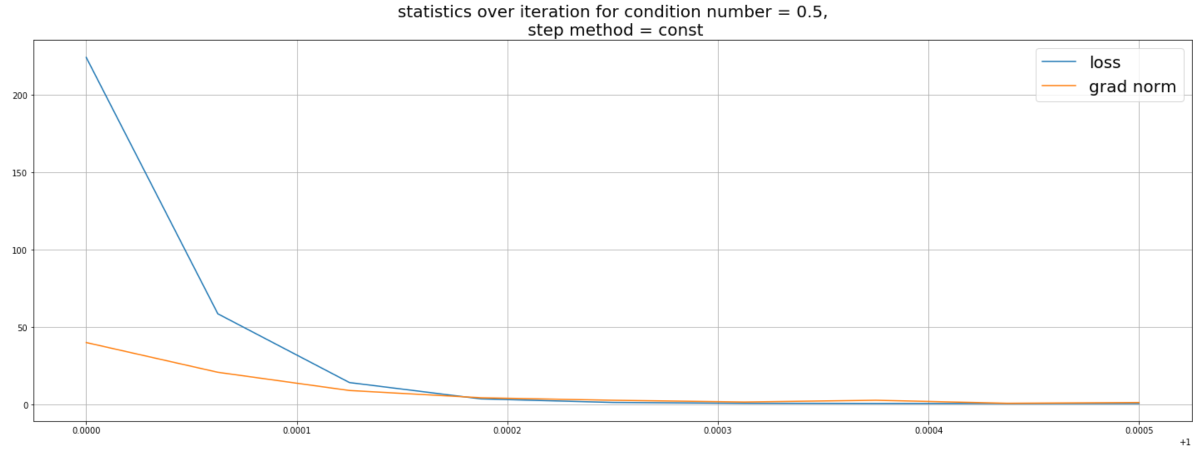


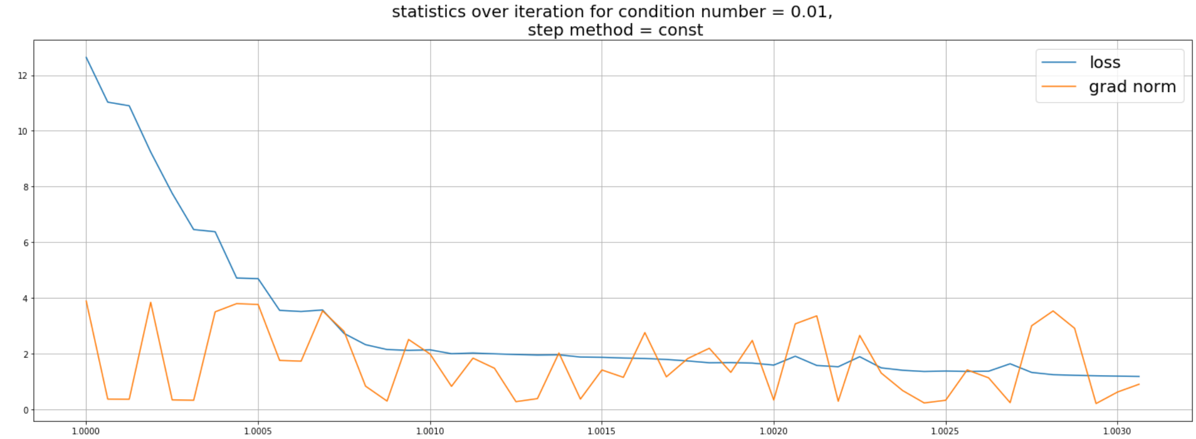
Stochastic gradient descent, different condition number comparison

In [42]:

```
plt.figure(figsize=(20, 15))
for i, conditionality_coef in enumerate([0.5, 0.01]):
    w, f_history, grad_history, weight = reg.fit(X, y, conditionality_coef=conditio
    plt.subplot(2, 1, i + 1)
    plot([w, w], [f_history, grad_history],
     'statistics over iteration for condition number = {},\n step method = {}'.forma
     labels=['loss', 'grad norm'])

plt.tight_layout()
plt.show()
```



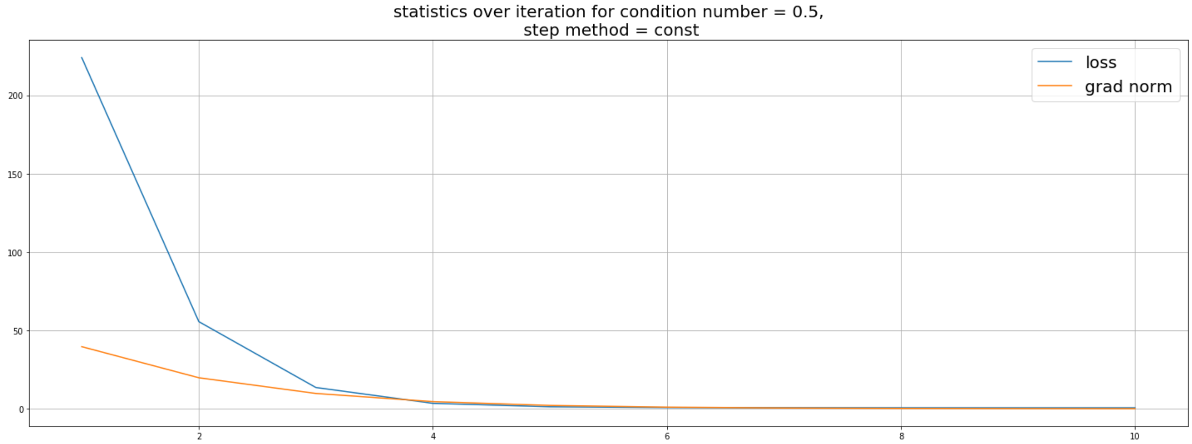


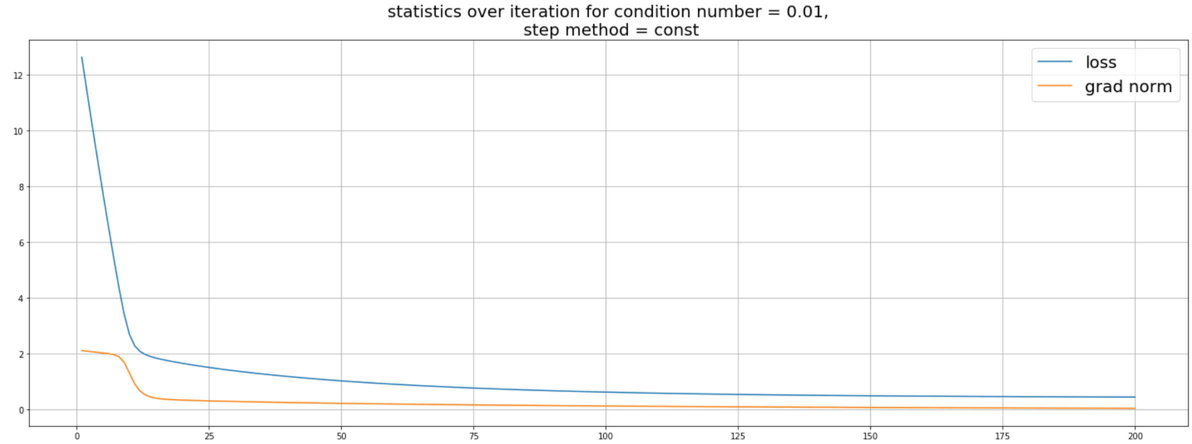
Nesterov momentum method, different condition number comparison

In [43]:

```
plt.figure(figsize=(20, 15))
for i, conditionality_coef in enumerate([0.5, 0.01]):
    w, f_history, grad_history, weight = reg.fit(X, y, conditionality_coef=conditio
    plt.subplot(2, 1, i + 1)
    plot([w, w], [f_history, grad_history],
     'statistics over iteration for condition number = {},\n step method = {}'.form
     labels=['loss', 'grad norm'])

plt.tight_layout()
plt.show()
```





Вывод:

> Стохастический градиентный спуск, несмотря на нестабильность `objective` сходится быстрее всех