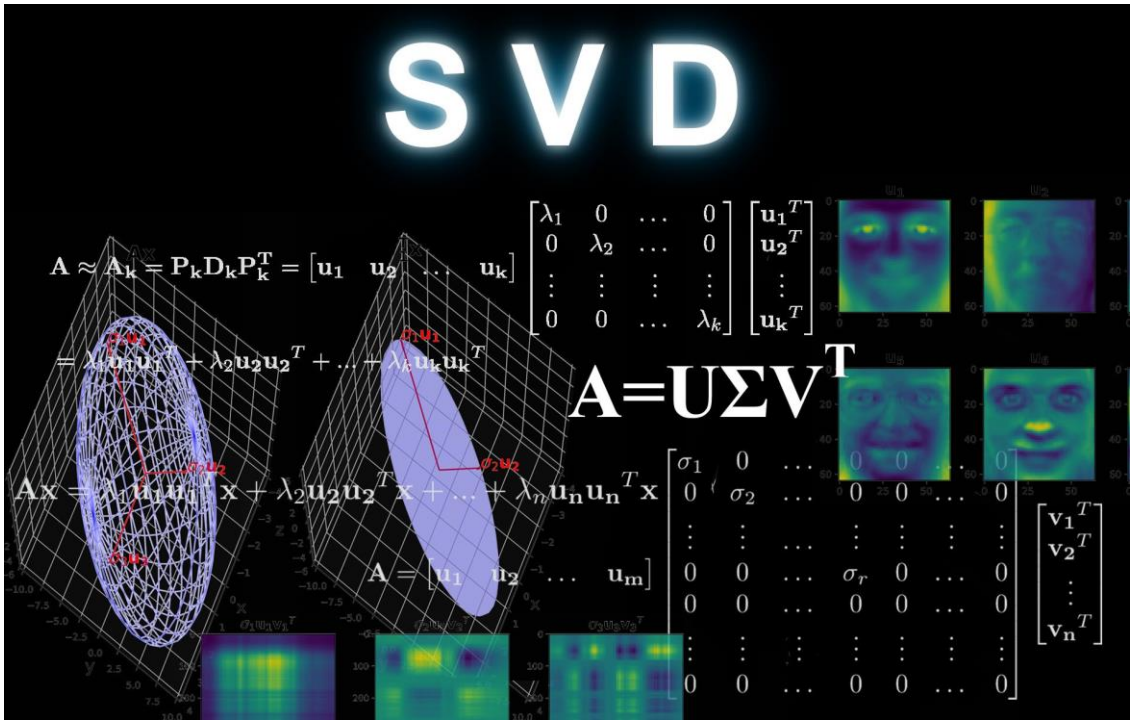
Универзитет „Св. Кирил и Методиј“ во Скопје
ФАКУЛТЕТ ЗА ИНФОРМАТИЧКИ НАУКИ И КОМПЈУТЕРСКО ИНЖЕНЕРСТВО

Универзитет „Св. Кирил и Методиј” – Скопје

Факултет за информатички науки и компјутерско инженерство (ФИНКИ)

Предмет: Линеарна алгебра

Тема: Singular Value Decomposition и нејзина примена



Студент: Филип Трајковски

Индекс: 171005

Ментори: д-р Жанета Попеска

д-р Марија Михова

Септември, 2020

1. ВОВЕД

Под поимот Singular Value Decomposition (SVD) се подразбира факторизација на матрица *A* од ред *m* × *n* во форма

$$A = U\,S\,V^T,$$

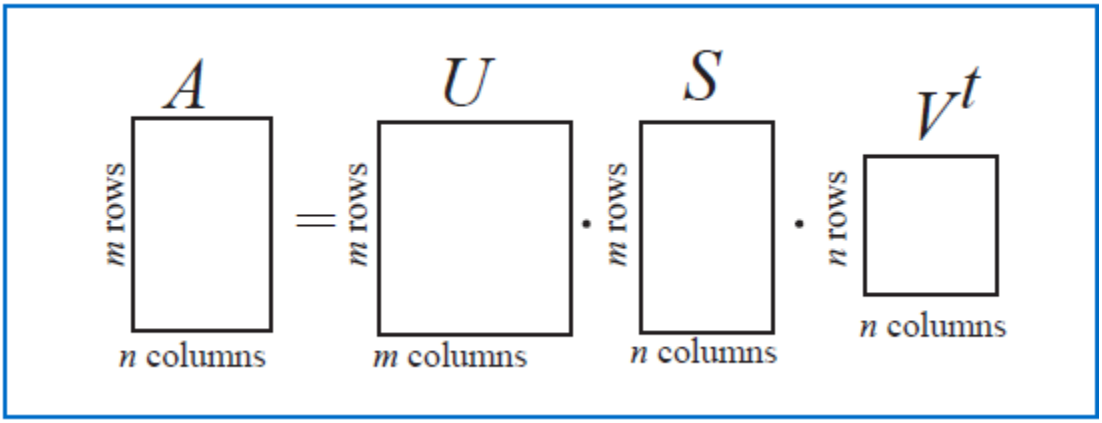
каде *U* е *m* × *m* ортогонална матрица, *V* е *n* × *n* ортогонална матрица, а *S* е *m* × *n* дијагонална матрица чиишто елементи различни од нула лежат на главната дијагонала. Понатаму, ќе претпоставиме дека $m \geq n$, и всушност при многу важни примени навистина m е многу поголемо од *n*.

Singular Value Decomposition има прилично долга историја, со тоа што првпат овој метод бил разгледуван од страна на математичарите во доцниот 19-ти век. Но, важните примени на оваа техника почекале сè додека пресметковната моќ на компјутерите не станала достапна во втората половина на 20-тиот век, кога беа развиени различни алгоритми за нејзина ефикасна имплементација.

2. ИЗВЕДУВАЊЕ НА SINGULAR VALUE DECOMPOSITION

Неквадратна матрица *A*, поточно матрица со различен број на редици и колони, не може да има сопствени вредности бидејќи *A***x** и **x** би биле вектори со различна димензија. Меѓутоа, постојат броеви коишто играат улога кај неквадратните матрици слична на онаа која ја имаат сопствените вредности кај квадратните матрици. Едно важно својство на Singular Value Decomposition за општа матрица од која било димензија е тоа што дозволува генерализација на сопствените вредности и сопствените вектори во ваква ситуација.

Целта на овој метод е да се определи факторизација на *m* × *n* матрицата *A*, каде што најчесто $m \geq n$, во формата $\mathrm{A} = U\,S\,V^T$. Притоа, *U* е *m* × *m* ортогонална матрица, *V* е *n* × *n* ортогонална матрица и *S* е *m* × *n* дијагонална матрица, таква што нејзините елементи различни од нула се $s_i \geq 0$, за $i = 1, \ldots, n$. Колоните на матрицата *U* во ваквата декомпозиција се нарекуваат леви сингуларни вектори на *A*, додека колоните на матрицата *V* се десни сингуларни вектори на *A*. Изгледот на добиените матрици при SVD е прикажан на следната слика.



Цената која што треба да ја платиме е тоа што ќе имаме две множества на сингуларни вектори, вектори **u** и вектори **v**. Векторите **u** се сопствени вектори на AA^T, а векторите **v** се сопствени вектори на A^TA. Бидејќи и двете матрици се симетрични, нивните сопствени вектори можат да се изберат така што ќе бидат ортонормални. Во равенката подолу, фактот дека $A(A^TA)$ е еднакво со $(AA^T)A$ води до извонредно својство за овие вектори **u** и **v**.

$$A\mathbf{v_1} = s_1\mathbf{u_1} \quad A\mathbf{v_2} = s_2\mathbf{u_2} \quad \dots \quad A\mathbf{v_k} = s_k\boldsymbol{u_k}$$

Сингуларните вектори $\mathbf{v_1}, \mathbf{v_2}, \dots, \mathbf{v_k}$ се наоѓаат во редичниот простор на *A*, додека пак векторите $\mathbf{u_1}, \mathbf{u_2}, \dots, \mathbf{u_k}$ се во просторот на колони на *A*. Сингуларните вредности $s_1, s_2, \dots, s_k$ се позитивни броеви и тие се сместуваат во дијагоналната матрица *S*. Кога векторите **v** и **u** се поставуваат како колони на матриците *V* и *U*, од нивната ортогоналност добиваме $V^TV = I$ и $U^TU = I$.

Но, потребни ни се уште $n - k$ вектори **v** и $m - k$ вектори **u** од нулевите простори $N(A)$ и $N(A^T)$. Овие множества вектори може да бидат ортонормални бази за овие два нулови простори, а со тоа автоматски ортогонални на првите *k* вектори **v** и **u**. Кога ќе ги вклучиме сите овие вектори **v** и **u**, матриците *V* и *U* стануваат квадратни. Новата матрица *S* е со димензии *m* × *n* бидејќи всушност на старата *k* × *k* матрица се додадени уште $m - k$ редици со нули и $n - k$ колони со нули.

Од равенките погоре, пресметувајќи колона по колона, добиваме дека $AV = US$. Бидејќи *V* е квадратна ортогонална матрица, нејзината инверзна матрица е всушност транспонираната $V^{-1} = V^T$. Па, од ова следува дека $A = U\,S\,V^T$, што претставува Singular Value Decomposition на матрицата *A*.

Во продолжение ќе биде детално објаснето формирањето на матриците *U*, *V* и *S*.

2.1. ФОРМИРАЊЕ НА МАТРИЦАТА S

Ја конструираме матрицата *S* преку наоѓање на сопствените вредности на *n* × *n* симетричната матрица A^TA. Сите овие сопствени вредности се ненегативни реални броеви и ги подредуваме од најголемиот кон најмалиот и ги означуваме со

$$s_1^2 \geq s_2^2 \geq \cdots \geq s_k^2 \geq s_{k+1} = \cdots = s_n = 0.$$

Тоа би значело дека со s_k^2 е означена најмалата сопствена вредност на A^TA различна од нула. Позитивните квадратни корени на сопствените вредности на A^TA ги даваат дијагоналните елементи на *S* и тие се наречени сингуларни вредности на *A*. Од овде,

$$S = \begin{bmatrix} s_1 & 0 & \cdots & 0 \\ 0 & s_2 & \ddots & \vdots \\ \vdots & \ddots & \ddots & 0 \\ 0 & \cdots & 0 & s_n \\ 0 & \cdots & \cdots & 0 \\ \vdots & & & \vdots \\ 0 & \cdots & \cdots & 0 \end{bmatrix},$$

каде $s_i = 0$, кога $k < i \leq n$.

Сингуларни вредности на *m* × *n* матрица *A* се позитивните квадратни корени на сопствените вредности на *n* × *n* симетричната матрица A^TA кои се различни од нула.

Кога *A* е симетрична *n* × *n* матрица, сите s_i^2 се сопствени вредности на $A^2 = A^TA$ и овие вредности ги претставуваат квадратите на сопствените вредности на *A*. Па затоа, во овој случај, сингуларните вредности се апсолутните вредности на сопствените вредности на *A*.

2.2. ФОРМИРАЊЕ НА МАТРИЦАТА V

Знаејќи дека *n* × *n* матрицата A^TA е симетрична, таа со сигурност има факторизација $A^TA = V\,D\,V^T$, каде *D* е дијагонална матрица чиишто дијагонални елементи се сопствените вредности на A^TA, *V* е ортогонална матрица таква што нејзината i-та колона е сопствениот вектор со l_2 норма еднаква на 1, кој одговара на сопствената вредност на i-тата дијагонална позиција на *D*. Специфичната дијагонална матрица зависи од редоследот на сопствените вредности долж дијагоналата. Па, го одбираме *D*, така што овие вредности се подредени во опаѓачки редослед. Колоните, означени со $\mathbf{v_1}, \mathbf{v_2}, \ldots, \mathbf{v_n}$, од *n* × *n* ортогоналната матрица

V се ортонормални вектори кои соодветствуваат на дадените сопствени вредности. Повеќекратните сопствени вредности на A^TA дозволуваат повеќе избори за соодветните сопствени вектори, па така, иако *D* е еднозначно определена, матрицата *V* може и да не биде. Но, тоа не е проблем, можеме да ја избереме било која матрица *V* според условите. Бидејќи сопствените вредности на A^TA се сите ненегативни, добиваме дека $D = S^2$.

2.3. ФОРМИРАЊЕ НА МАТРИЦАТА U

Знаеме дека ненултите сопствени вредности на A^TA и оние на AA^T се исти. Дополнително, соодветните сопствени вектори на симетричните матрици A^TA и AA^T формираат комплетни ортонормални подмножества на R^n и R^m, соодветно. Па затоа, ортонормалното множество од *n* сопствени вектори за A^TA ги формира колоните на *V*, а пак ортонормалното множество од *m* сопствени вектори за AA^T ги формира колоните на *U* на истиот начин.

Иако изведувањето на SVD е јасно теоретски, сепак во пракса, не е паметно да се прави спектрална декомпозиција на матрицата AA^T бидејќи истата може да биде со огромни димензии *m* × *m* со што би зафаќала голем мемориски простор, а исто така и би не чинело значително време. Затоа и за наоѓање на матрицата *U* ќе ја искористиме A^TA, која што е со помали димензии. Во продолжение е опишан начинот со кој се подобрува временската и просторната сложеност избегнувајќи пресметки врз AA^T, за кои што е потребно $O(m^3)$ време и $O(m^2)$ простор.

За да ја конструираме *m* × *m* матрицата *U*, најпрво ги разгледуваме вредностите $s_1 \geq s_2 \geq \cdots \geq s_k > 0$ и соодветните колони во *V* дадени со векторите $\mathbf{v_1}, \mathbf{v_2}, \ldots, \mathbf{v_k}$. Дефинираме

$$\mathbf{u_i} = \frac{1}{s_i} A\mathbf{v_i}, \qquad \text{за } i = 1, 2, \ldots, k.$$

Ги користиме овие вектори како први *k* колони од вкупно *m* колони на *U*. Бидејќи A е *m* × *n* матрица и секој од векторите $\mathbf{v_i}$ е *n* × *1*, векторите $\mathbf{u_i}$ се *m* × *1*, како што е потребно. Дополнително, за секој $1 \leq i \leq k$ и $1 \leq j \leq k$, од фактот што векторите $\mathbf{v_1}, \mathbf{v_2}, \ldots, \mathbf{v_n}$ се сопствени вектори на A^TA кои формираат ортонормално множество следува дека

$$\mathbf{u}_i^t \mathbf{u}_j = \left(\frac{1}{s_i} A\mathbf{v}_i\right)^t \frac{1}{s_j} A\mathbf{v}_j = \frac{1}{s_i s_j} \mathbf{v}_i^t A^t A \mathbf{v}_j = \frac{1}{s_i s_j} \mathbf{v}_i^t s_j^2 \mathbf{v}_j = \frac{s_j}{s_i} \mathbf{v}_i^t \mathbf{v}_j = \begin{cases} 0 & \text{if } i \neq j, \\ 1 & \text{if } i = j. \end{cases}$$

Па така, првите *k* колони на *U* формираат ортонормално множество вектори во R^m. Меѓутоа, ни требаат дополнителни $m - k$ колони за *U*. За ова, прво е потребно да најдеме $m - k$ вектори такви што кога ќе се додадат на векторите од првите *k* колони ќе се добие линеарно независно множество вектори. Тогаш, ќе можеме да го примениме процесот на Грам-Шмит за да ги добиеме преостанатите колони.

Матрицата *U* нема да биде уникатна освен ако $k = m$ и тоа само ако сопствените вредности на $A^T A$ се уникатни. Оваа неединственост не е никаков проблем бидејќи нам ни е потребна само една таква матрица *U*, која ќе биде соодветно избрана.

2.4. ДОКАЗ

За да потврдиме дека овој процес навистина ја дава посакуваната факторизација $A = U\,S\,V^T$, прво ќе се навратиме на фактот дека транспонираната матрица на ортогонална матрица е исто така и инверзна на самата матрица. Затоа, наместо да докажеме дека $A = U\,S\,V^T$, ние ќе го покажеме еквивалентното тврдење $A\,V = U\,S$.

Векторите $\mathbf{v_1}, \mathbf{v_2}, \ldots, \mathbf{v_n}$ формираат база за R^n, $A\mathbf{v_i} = s_i\mathbf{u_i}$ за $i = 1, 2, \ldots, k$ и $A\mathbf{v_i} = \mathbf{0}$ за $i = k+1, \ldots, n$. Само првите *k* колони на *U* произведуваат ненулти вредности во производот US, па добиваме

$$
\begin{aligned}
AV &= A\begin{bmatrix}\mathbf{v}_1 & \mathbf{v}_2 & \cdots & \mathbf{v}_k & \mathbf{v}_{k+1} & \cdots & \mathbf{v}_n\end{bmatrix} \\
&= \begin{bmatrix}A\mathbf{v}_1 & A\mathbf{v}_2 & \cdots & A\mathbf{v}_k & A\mathbf{v}_{k+1} & \cdots & A\mathbf{v}_n\end{bmatrix} \\
&= \begin{bmatrix}s_1\mathbf{u}_1 & s_2\mathbf{u}_2 & \cdots & s_k\mathbf{u}_k & \mathbf{0} & \cdots & \mathbf{0}\end{bmatrix} \\
&= \begin{bmatrix}\mathbf{u}_1 & \mathbf{u}_2 & \cdots & \mathbf{u}_k & \mathbf{0} & \cdots & \mathbf{0}\end{bmatrix}
\begin{bmatrix}
s_1 & 0 & \cdots & 0 & 0 & \cdots & 0 \\
0 & \ddots & \ddots & \vdots & \vdots & & \vdots \\
\vdots & \ddots & \ddots & \vdots & \vdots & & \vdots \\
0 & \cdots & 0 & s_k & 0 & \cdots & 0 \\
0 & \cdots & \cdots & 0 & 0 & \cdots & 0 \\
\vdots & & & \vdots & \vdots & & \vdots \\
0 & \cdots & \cdots & 0 & 0 & \cdots & 0
\end{bmatrix} = US,
\end{aligned}
$$

Со ова се завршува формирањето на Singular Value Decomposition на матрицата *A*.

2.5. КРАТКО РЕЗИМЕ

Како заклучок, за да се определи Singular Value Decomposition на *m* × *n* матрицата A, треба:

- Да се пронајдат сопствените вредности $s_1^2 \geq s_2^2 \geq \cdots \geq s_k^2 \geq s_{k+1} = \cdots = s_n = 0$ за симетричната матрица $A^T A$ и да се постават позитивните квадратни корени на s_i^2 на позиција s_{ii} во *m* × *n* дијагоналната матрица S.

- Да се најде множество на ортонормални сопствени вектори $\{\mathbf{v_1}, \mathbf{v_2}, \dots, \mathbf{v_n}\}$ соодветни на сопствените вредности на A^TA и да се конструира $n \times n$ матрицата V со овие вектори како нејзини колони.
- Да се најде множество на ортонормални сопствени вектори $\{\mathbf{u_1}, \mathbf{u_2}, \dots, \mathbf{u_m}\}$ соодветни на сопствените вредности на AA^T и да се конструира $m \times m$ матрицата U со овие вектори како нејзини колони.

3. АНАЛИЗА НА ВРЕМЕНСКАТА И ПРОСТОРНАТА СЛОЖЕНОСТ НА SVD

За анализа на временската комплексност на алгоритмот ќе го користиме бројот на FLoating-point OPerations (FLOP) како мерка. За матрици $A \in R^{m\times n}$ и $B \in R^{n\times l}$, временската комплексност на нивниот производ AB побарува mnl FLOP за производите и $ml(n-1)$ FLOP за збировите. Оттука, за производот на две матрици потребни ни се $O(mnl + ml(n-1)) = O(2mnl - ml) = O(mnl)$ FLOP. Аналогно на оваа пресметка, ќе бидат изведени временската и просторната сложеност на SVD со претходно опишаниот поефикасен метод, како и споредба со изведувањето на SVD користејќи ја матрицата AA^T.

3.1. ВРЕМЕНСКА СЛОЖЕНОСТ НА SVD

1. За формирањето на матрицата $A^TA \in R^{n\times n}$ потребни се $O(mn^2)$ FLOP.

2. За изведувањето на декомпозицијата на A^TA со помош на сопствените вредности (наоѓање на матрицата V) потребни се $O(n^3)$ FLOP.

3. За пресметувањето на квадратен корен за секоја сопствена вредност на A^TA потребни се $O(n)$ FLOP.

4. За формирањето на векторите $\mathbf{u_i} = \frac{1}{s_i}A\mathbf{v_i}$ потребни се $O(n(mn+m))$ FLOP бидејќи за пресметување на производот $A\mathbf{v_i}$ се потребни $O(mn)$ FLOP, додека пак за делењето со s_i потребни се $O(m)$ FLOP. Вкупно имаме n вакви равенки, па затоа имаме $O(n(mn+m))$ FLOP.

Вкупната временска сложеност за изведување на Singular Value Decomposition е $O(2mn^2 + n^3 + n + mn)$. Бидејќи при практична примена на овој алгоритам m > n, сложеноста во таквиот случај би била $O(mn^2)$, а доколку $m = n$ би била $O(m^3) = O(n^3)$.

Доколку во чекорот 4. ја пресметувавме матрицата AA^T би имале сложеност од $O(m^2n)$ со што лесно се воочува бенефитот од претходната постапка ако се земе предвид дека во пракса најчесто $m > n$.

3.2. ПРОСТОРНА СЛОЖЕНОСТ НА SVD

Можеме да ја пресметаме просторната сложеност на овој алгоритам според бројот на елементите во матриците. За матрица A со димензии *m* × *n,* велиме дека нејзината просторна комплексност е $O(mn)$.

1. За формирањето на матрицата $A^TA \in R^{n\times n}$ потребно е $O(n^2)$.

2. За наоѓањето на сопствените вредности и вектори на $A^TA \in R^{n\times n}$ потребно е $O(n^2+n)$.

3. За пресметувањето на квадратните корени на секоја сопствена вредност на A^TA потребно е $O(n)$.

Ова значи дека за чување на матрицата *V* ни е потребно $O(n^2)$, додека за чување на *n*-те сингуларни вредности (матрицата S) имаме $O(n)$.

4. За формирањето на матрицата U (поточно секој од векторите $\mathbf{u_i} = \frac{1}{s_i}A\mathbf{v_i}$) потребно е $O(m^2)$ бидејќи секоја вектор-колона $\mathbf{u_i}$ има *m* елементи и имаме *m* вакви равенки.

Вкупната просторна комплексност за SVD, земајќи предвид дека за чување на матрицата *A* ни треба $O(mn)$, би била $O(2n^2+2n+m^2+mn)$. Бидејќи најчесто при практична употреба на овој алгоритам $m \geq n$, просторната сложеност би била $O(m^2)$.

4. ИМПЛЕМЕНТАЦИЈА НА SINGULAR VALUE DECOMPOSITION ВО ПРОГРАМСКИОТ ЈАЗИК PYTHON

Во продолжение ќе биде дадена една имплементација на SVD алгоритмот во код во програмскиот јазик Python, а потоа ќе бидат разгледани и две готови функции за пресметување на матриците од оваа декомпозиција кои се дел од библиотеките во Python.

```
 8 def randomUnitVector(n):
 9     unnormalized = [normalvariate(0, 1) for _ in range(n)]
10     theNorm = sqrt(sum(x * x for x in unnormalized))
11     return [x / theNorm for x in unnormalized]
```

```python
def svd_1d(A, epsilon=1e-10):
    ''' The one-dimensional SVD '''

    n, m = A.shape
    x = randomUnitVector(min(n,m))
    lastV = None
    currentV = x

    if n > m:
        B = np.dot(A.T, A)
    else:
        B = np.dot(A, A.T)

    iterations = 0
    while True:
        iterations += 1
        lastV = currentV
        currentV = np.dot(B, lastV)
        currentV = currentV / norm(currentV)

        if abs(np.dot(currentV, lastV)) > 1 - epsilon:
            print("converged in {} iterations!".format(iterations))
            return currentV
```

```python
def svd(A, k=None, epsilon=1e-10):
    '''
        Compute the singular value decomposition of a matrix A
        using the power method. A is the input matrix, and k
        is the number of singular values you wish to compute.
        If k is None, this computes the full-rank decomposition.
    '''
    A = np.array(A, dtype=float)
    n, m = A.shape
    svdSoFar = []
    if k is None:
        k = min(n, m)

    for i in range(k):
        matrixFor1D = A.copy()

        for singularValue, u, v in svdSoFar[:i]:
            matrixFor1D -= singularValue * np.outer(u, v)

        if n > m:
            v = svd_1d(matrixFor1D, epsilon=epsilon)  # next singular vector
            u_unnormalized = np.dot(A, v)
            sigma = norm(u_unnormalized)  # next singular value
            u = u_unnormalized / sigma
        else:
            u = svd_1d(matrixFor1D, epsilon=epsilon)  # next singular vector
            v_unnormalized = np.dot(A.T, u)
            sigma = norm(v_unnormalized)  # next singular value
            v = v_unnormalized / sigma
```

```
        svdSoFar.append((sigma, u, v))

    singularValues, us, vs = [np.array(x) for x in zip(*svdSoFar)]
    return singularValues, us.T, vs
```

```
if __name__ == "__main__":
    movieRatings = np.array([
        [2, 5, 3],
        [1, 2, 1],
        [4, 1, 1],
        [3, 5, 2],
        [5, 3, 1],
        [4, 5, 5],
        [2, 4, 2],
        [2, 2, 5],
    ], dtype='float64')

    # v1 = svd_1d(movieRatings)
    # print(v1)

    theSVD = svd(movieRatings)
```

За среќа, наместо оваа имплементација може да ги искористиме готовите имплементации на SVD кои ни ги нудат библиотеките NumPy и scikit-learn и истите се далеку побрзи и поефикасни.

NumPy е фундаментална библиотека за пресметување за научни цели во Python. Покрај другите можности кои ги нуди, таа исто така содржи имплементации на многу методи од линеарна алгебра. Во конкретниот случај кога станува збор за SVD, декомпозицијата на матрицата во матриците *U*, *S* и *V* може да ја добиеме со користење на numpy.linalg функцијата. *S* е дијагонална матрица, што значи дека најголем дел од нејзините елементи се еднакви на нула. Ваквите матрици се нарекуваат ретки (sparse) матрици, па за да се зачува мемориски простор истата се враќа како еднодимензионална низа од сингуларни вредности наместо како комплетна дводимензионална матрица. Пример за нејзино користење е даден во прилог.

```python
import numpy as np
from numpy.linalg import svd

# define your matrix as a 2D numpy array
A = np.array([[4, 0], [3, -5]])

U, S, VT = svd(A)

print("Left Singular Vectors:")
print(U)
print("Singular Values:")
print(np.diag(S))
print("Right Singular Vectors:")
print(VT)

print(U @ np.diag(S) @ VT)
```

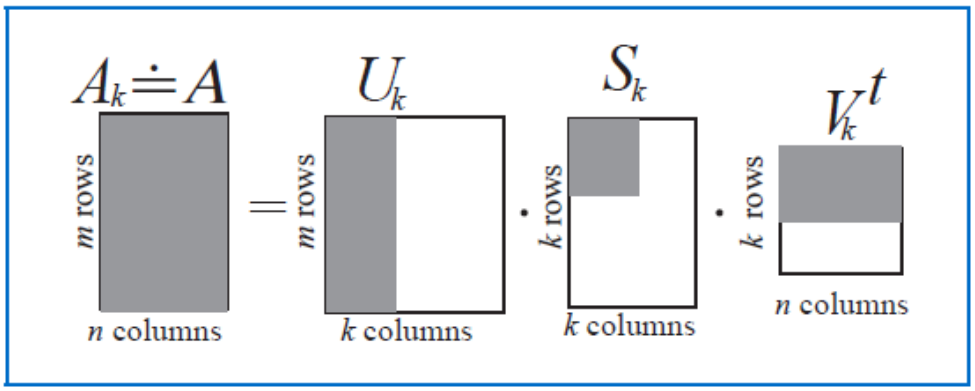
Во најголем дел од практичните примени не ни се потребни целосните матрици *U*, *S* и *V*. Во тој случај, подобро е да се искористи методот Truncated SVD кој наместо да ја најде целосната декомпозиција на матрицата, прави нејзина апроксимација користејќи ги само првите *k* сингуларни вредности. Бидејќи овој метод е доста побрз во споредба со обичната SVD и дава одлични резултати, истиот се применува почесто и затоа ќе биде искористен кога ќе ги разгледуваме апликациите на SVD. За оваа цел ќе ја искористиме функцијата sklearn.decomposition од библиотеката scikit-learn. Притоа, се специфицираат бројот на карактеристики кои ги сакаме како излез со помош на n_components параметарот. Секако вредноста на овој параметар мора да биде помала од бројот на атрибути/карактеристики на влезната матрица. Во продолжение е дадена неговата примена на едноставна матрица.

```python
import numpy as np
from sklearn.decomposition import TruncatedSVD

A = np.array([[-1, 2, 0], [2, 0, -2], [0, -2, 1]])
print("Original Matrix:")
print(A)

svd =  TruncatedSVD(n_components = 2)
A_transf = svd.fit_transform(A)

print("Singular values:")
print(svd.singular_values_)

print("Transformed Matrix after reducing to 2 features:")
print(A_transf)
```

Во следниот дел овие имплементации ќе бидат искористени при обработка на реални примери на примена на SVD, особено осврнувајќи се на примената при компресија на слики.

5. ПРИМЕНА НА SINGULAR VALUE DECOMPOSITION

Причината за важноста на оваа декомпозиција на матрица во многу практични примени лежи во тоа што дозволува да се најдат најважните карактеристики на $m \times n$ матрица користејќи притоа матрица која најчесто е со значително помали димензии. Бидејќи сингуларните вредности се излистани по дијагоналата на S во опаѓачки редослед, оставањето на само првите k колони и редици од S ја произведува најдобрата возможна апроксимација во овие димензии на матрицата A. Па така, ја заменуваме матрицата S со $k \times k$ матрицата S_k која што ги содржи само најзначајните сингуларни вредности. Овие ќе бидат со сигурност тие што се различни од нула, но исто така можеме да ги избришеме и сингуларните вредности кои што се релативно мали (блиски до нула). Понатаму ги определуваме соодветните $m \times k$ и $k \times n$ матрици U_k и V_k^T, во согласност со претходната процедура. Овој метод уште се нарекува и truncated (скратена) SVD и истиот е илустративно покажан на следната слика.



Потоа, новата матрица $A_k = U_k S_k V_k^T$ е сè уште со димензии $m \times n$ и ќе побарува $O(mn)$ простор за нејзина репрезентација. Но, во факторизираната форма, потребниот простор за чување на податоците е $O(mk)$ за U_k, $O(k)$ за S_k и $O(nk)$ за V_k^T или вкупно $O(k(m+n+1))$.

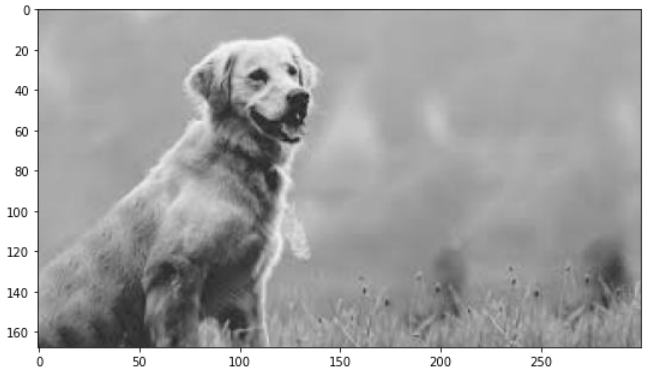
Да претпоставиме дека, на пример, $m = 2n$ и $k = n/3$. Тогаш, оригиналната матрица A содржи $mn = 2n^2$ податоци. Факторизацијата која ја произведува A_k содржи само $mk = 2n^2/3$ податоци за U_k, k за S_k и $nk = n^2/3$ за V_k^T што вкупно претставува $(n/3)(3n^2+1)$. Ова е редукција од речиси 50% од првобитниот потребен простор за да се смести матрицата A и резултира во одличен начин на компресирање на податоци.

Во продолжение ќе биде прикажан практичен пример на примена на SVD при компресија на слики користејќи го програмскиот јазик Python. Компресијата на слики го користи фактот што само мал дел од сингуларните вредности добиени со SVD се големи, па на тој начин лесно може да се апроксимира оригиналната слика користејќи значително помалку меморија. Исто така, на овој начин може да се добијат слики кои што човечкото око тешко може да ги разликува од оригиналот.

Во нашиот пример ќе ја разгледаме сликата во прилог. Најпрво ќе ја конвертираме во црно-бела слика со што понатаму може да ја гледаме како матрица чиишто броеви го претставуваат интензитетот на соодветниот пиксел и потоа ќе направиме Singular Value Decomposition на истата.

```
In [2]: img = Image.open('dog_photo.jpg')
        # convert image to grayscale
        imggray = img.convert('LA')
        plt.figure(figsize=(9, 6))
```

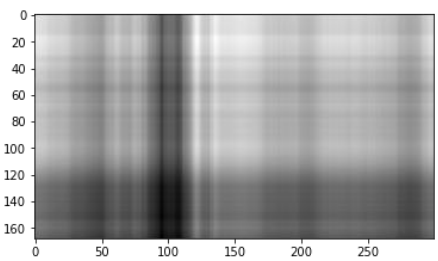
```
In [3]: # convert to numpy array
        imgmat = np.array(list(imggray.getdata(band=0)), float)
        # Reshape according to orginal image dimensions
        imgmat.shape = (imggray.size[1], imggray.size[0])
        imgmat = np.matrix(imgmat)
        plt.figure(figsize=(9,6))
        plt.imshow(imgmat, cmap='gray');
```

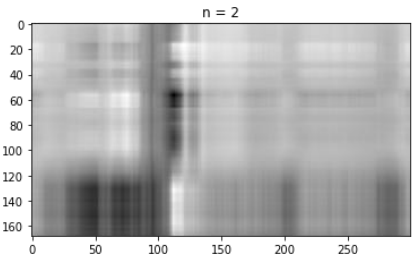


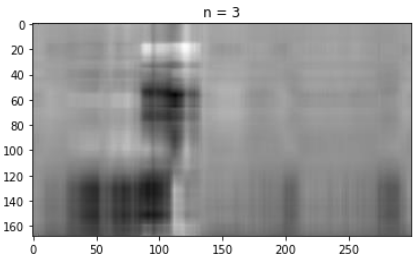

```
In [4]: U, sigma, V = np.linalg.svd(imgmat)
```

```
In [5]: reconstimg = np.matrix(U[:, :1]) * np.diag(sigma[:1]) * np.matrix(V[:1, :])
        plt.imshow(reconstimg, cmap='gray');
```

Пресметувањето на апроксимацијата користејќи ја првата колона на *U* и првата редица на *V* ја репродуцира најистакнатата карактеристика на сликата. Знаејќи дека првите неколку сингуларни вектори и сингуларни вредности објаснуваат најголем дел од варијансата, со нив успешно би можеле да направиме реконструкција на главните елементи на сликата.

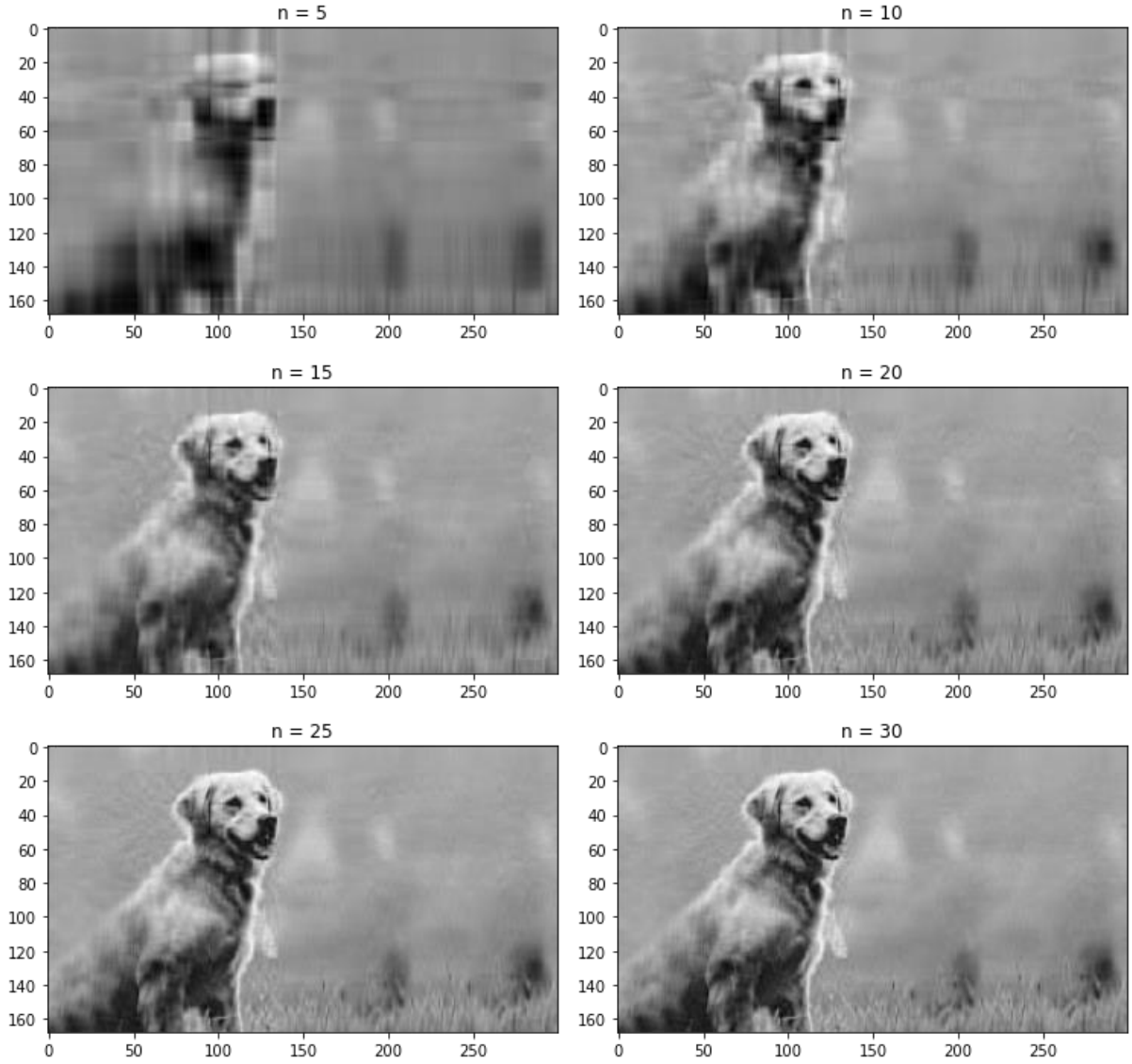


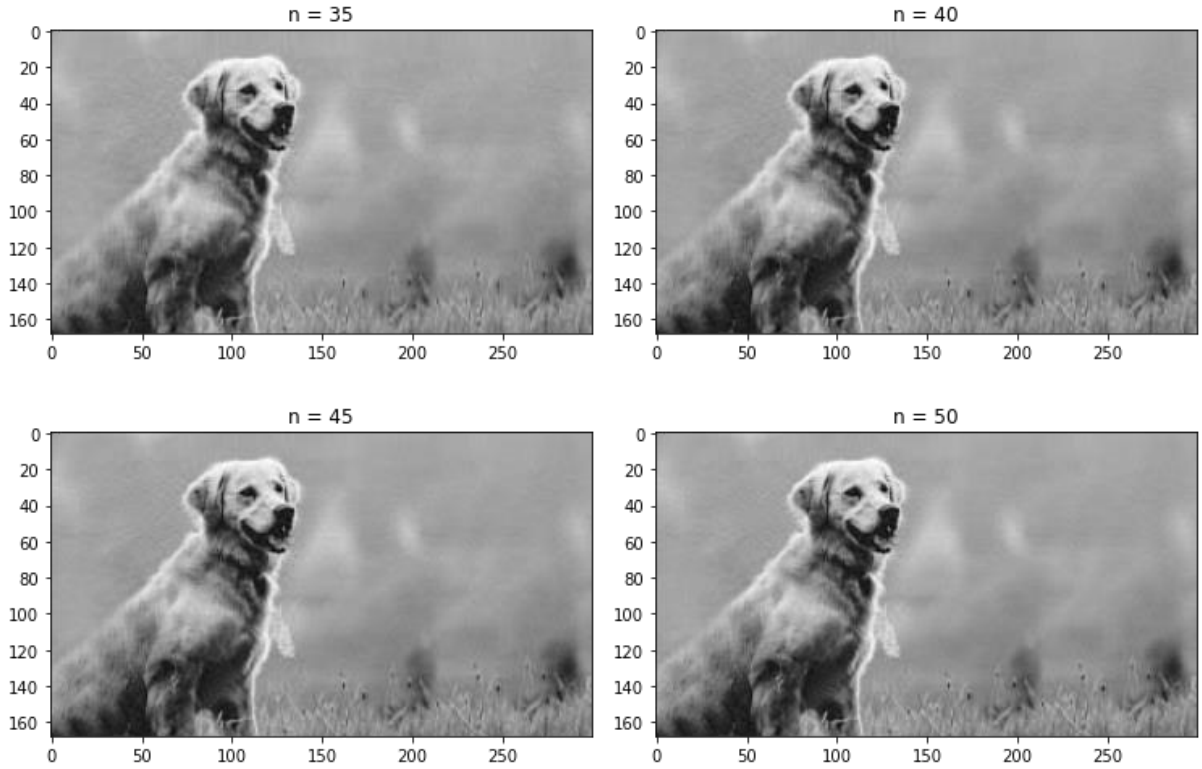




Погоре се прикажани добиените компресирани слики користејќи ги најголемите една, две и три сингуларни вредности соодветно. Секако дека само со две или три најзначајни карактеристики не сме ни приближно блиску до нашата слика. Затоа понатаму ќе ја реконструираме сликата користејќи различен број сингуларни вредности и притоа ќе воочиме како таа сè повеќе наликува на оригиналната со зголемување на бројот на искористени сингуларни вектори. Првите 40 сингуларни вредности произведуваат слика која што е многу слична на оригиналната и понатамошните слики реконструирани со повеќе сингуларни вредности незначително се подобруваат.

```
In [7]: for i in range(5, 51, 5):
            reconstimg = np.matrix(U[:, :i]) * np.diag(sigma[:i]) * np.matrix(V[:i, :])
            plt.imshow(reconstimg, cmap='gray')
            title = "n = %s" % i
            plt.title(title)
            plt.show()
```





Доколку сметаме дека сме задоволни со квалитетот на сликата (чии димензии се 168 x 300) добиена со првите 40 сингуларни вредности, тогаш потребниот мемориски простор за нејзино зачувување би бил $\frac{40*168+40+40*300}{168*300} \approx 37.2\%$ од потребната меморија за оригиналната црно-бела слика што претставува навистина добра заштеда.

Ќе разгледаме уште еден пример на компресија на слика, но овој пат со поголеми димензии (3840 x 2160) и без да ја трансформираме во црно-бела боја, со што ќе направиме споредба на применливоста на SVD во поглед на добиениот квалитет и брзина во различни ситуации.

Како и во претходниот пример, откако ќе ја вчитаме сликата, истата ја трансформираме во NumPy низа од пиксели. Потоа ги селектираме црвениот, зелениот и синиот канал на бои од сликата.

```
def load_image(image):
    image = Image.open(image)
    im_array = numpy.array(image)

    red = im_array[:, :, 0]
    green = im_array[:, :, 1]
    blue = im_array[:, :, 2]

    return red, green, blue
```

Откако ги раздвоивме трите бои од RGB просторот, сега ќе извршиме компресија со помош на SVD на секој од каналите со користење на соодветната функција од NumPy. Потоа, ќе креираме низа од нули која ќе ја пополниме по множењето на добиените матрици. Исто така, го специфицираме и лимитот на сингуларни вредности до кој сакаме да ги извршиме пресметките. Откако ќе ги помножиме добиените матрици *U*, *S* и *V* ги добиваме компресираните вредности и ги трансформираме во uint8 податочниот тип.

In [3]:
```
def channel_compress(color_channel, singular_value_limit):
    u, s, v = numpy.linalg.svd(color_channel)
    compressed = numpy.zeros((color_channel.shape[0], color_channel.shape[1]))
    n = singular_value_limit

    left_matrix = numpy.matmul(u[:, 0:n], numpy.diag(s)[0:n, 0:n])
    inner_compressed = numpy.matmul(left_matrix, v[0:n, :])
    compressed = inner_compressed.astype('uint8')
    return compressed

red, green, blue = load_image("nature.jpg")
singular_val_lim = 1000
```

На крај, ги спојуваме трите канали на бои заедно и ја прикажуваме новодобиената слика.

In [4]:
```
def compress_image(red, green, blue, singular_val_lim):
    compressed_red = channel_compress(red, singular_val_lim)
    compressed_green = channel_compress(green, singular_val_lim)
    compressed_blue = channel_compress(blue, singular_val_lim)

    im_red = Image.fromarray(compressed_red)
    im_blue = Image.fromarray(compressed_blue)
    im_green = Image.fromarray(compressed_green)

    new_image = Image.merge("RGB", (im_red, im_green, im_blue))
    new_image.show()
    new_image.save("nature-edited.jpg")

compress_image(red, green, blue, singular_val_lim)
```

Во случајов, направивме три компресирани слики, со првите 600, 800 и 1000 сингуларни вредности. На првата слика добиена со користење на првите 600 сингуларни вектори се забележува одреден шум на некои делови, но последната слика добиена со поставен лимит од 1000 вредности е многу слична со оригиналната и е со навистина задоволителен квалитет.

Оригинална слика

n = 600

n = 800

n = 1000

Секако дека во споредба со претходната слика која беше со далеку помали димензии и црно-бела, алгоритмот се извршуваше значително побавно на сликата со поголеми димензии, но крајните резултати покажуваат дека и во двата случаи можеме да извршиме огромно ниво на компресија со помош на truncated SVD и притоа да добиеме одличен квалитет.

КОРИСТЕНА ЛИТЕРАТУРА

1. Richard L. Burden, J. Douglas Faires (2011). *Numerical analysis*. (9th ed.) Boston, MA: Cengage Learning.
2. David C. Lay (2012). *Linear algebra and its applications* (4th ed.). Reading, Mass.: Addison-Wesley.
3. Strang, G. (2009). *Introduction to linear algebra* (4th ed.). Wellesley (Massachusetts): Wellesley-Cambridge Press.
4. X. Li, S. Wang, Y. Cai (2019). *Tutorial: Complexity analysis of Singular Value Decomposition and its variants,* Institute of Automation, Chinese Academy of Sciences, University of Chinese Academy of Sciences
   Tutorial: Complexity analysis of Singular Value Decomposition and its variants
5. Computation of the Singular Value Decomposition
6. https://hadrienj.github.io/posts/Deep-Learning-Book-Series-2.8-Singular-Value-Decomposition/
7. https://www.frankcleary.com/svdimage/
8. https://www.analyticsvidhya.com/blog/2019/08/5-applications-singular-value-decomposition-svd-data-science/
9. https://stackabuse.com/dimensionality-reduction-in-python-with-scikit-learn/
10. https://github.com/j2kun/svd