САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

КАФЕДРА ТЕХНОЛОГИИ ПРОГРАММИРОВАНИЯ

Токарев Роман Андреевич

Выпускная квалификационная работа бакалавра

Метод идентификации веществ по их спектрам

Направление 010400

Прикладная математика и информатика

Научный руководитель,

старший преподаватель

Стученков А. Б.

Санкт-Петербург

2016

Содержание

Введение

Автоматизированный анализ данных имеет множество достоинств, в связи с чем используется практически во всех областях, которыми занимается человек. Объем данных становится слишком большим для целостного восприятия человеком, потому разработка и внедрение алгоритмов, анализирующих данные или упрощающих их восприятие, становятся актуальной задачей повсеместно.

Спектральный анализ (или спектроскопия) — это набор методов, применяемых с целью обнаружения и определения количества элементов, радикалов и соединений, которые входят в состав объекта. Эти методы основываются на изучении спектров воздействия излучения на материю.

Спектроскопия довольно давно и широко используется в химии, физике и астрономии, в большинстве случаев позволяя эффективно идентифицировать вещество или объект и его состав по результатам анализа графика спектра в различных световых диапазонах.

Из преимуществ спектрального анализа можно отметить:

- анализ небольшого количества материала (около 0,1 грамма);
- анализ без разрушения объекта;
- универсальность методов анализа;
- высокая точность;
- высокая производительность.

В данной работе решается задача поиска метода для идентификации породы образцов древесины по их спектрам, так как идентификация материала со сложным составом (например, древесины) является непростой задачей, и, в случае её успешного решения, данный метод можно распространить на другие объекты со сложным составом.

Существенной сложностью в задаче идентификации по спектрам являются проблемы в процессе непосредственного сбора спектров [2]:

- шумы и помехи, связанные с неидеальными компонентами прибора;

- воздействие внешних факторов, таких как температура, освещённость и пр.;
- гетерогенность образцов ввиду естественных причин и человеческого фактора.

Классифицируемые множества обрабатываемых данных *не являются линейно сепарабельными*, в связи с чем стандартные методы классификации, такие как *k ближайших соседей* (k-nearest neighbors) или *k взвешенных ближайших соседей*, не дают хорошего результата. Для решения этой задачи была использована нейронная сеть, устроенная по типу *многослойного персептрона* (multilayer perceptron) (также в русскоязычной литературе встречается «перцептрон») [3], для предварительной подготовки данных применялся *метод главных компонент* (Principal Component Analysis, PCA) [4] и *линейный дискриминантный анализ* (Linear Discriminant Analysis, LDA) [5].

В начале работы осуществляется постановка задачи и оценка существующих методов спектрального анализа.

В первой главе описываются использованные методы предварительной обработки данных.

Вторая глава посвящена описанию и практической реализации нейронной сети по типу многослойного персептрона, а также методу её обучения.

Третья глава, которая является последней, посвящена подбору оптимальных параметров и оценке полученных результатов.

Постановка задачи

Для проведения исследования использовались поточечные данные спектров 604 образцов четырнадцати пород древесины, полученные с применением различных типов светового излучения:

- видимое излучение в диапазоне 242–800 нм (2048 точек);
- инфракрасное в диапазоне 1094–2032 нм (256 точек);
- светодиодное излучение в диапазоне 322–1095 нм (2048 точек).

Выборка неоднородна, поскольку в ней имеются «выбросы», шумы, а также нередки случаи, когда спектры образуют чётко наблюдаемые подгруппы внутри одной породы (см. Рис. 1, Рис. 2).

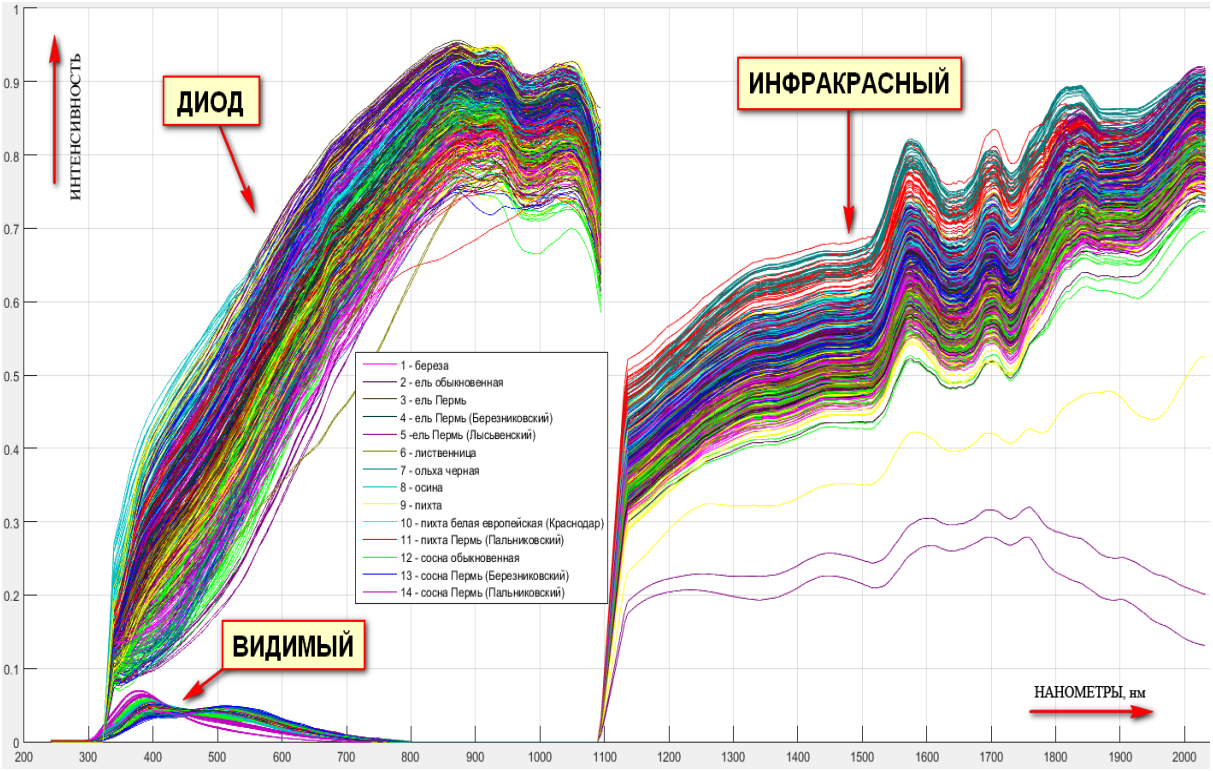


Рис. 1. Исходные данные спектров.

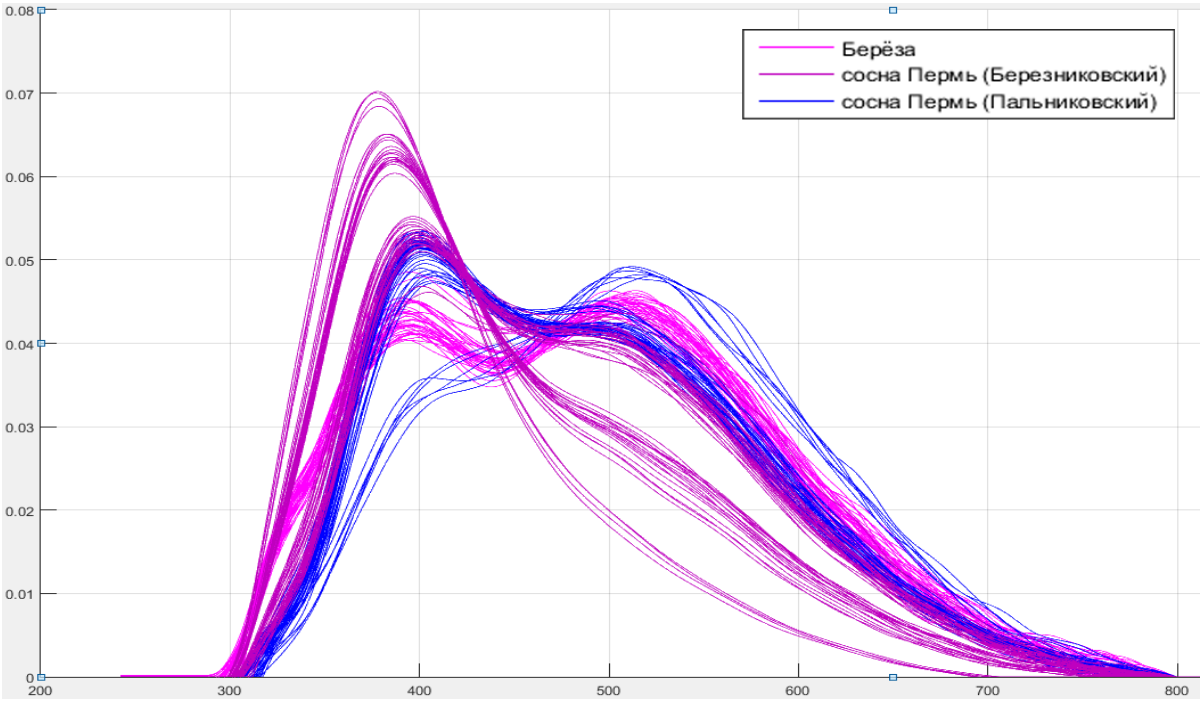


Рис. 2. Графики спектров трёх пород древесины в видимом излучении.

Целью данной работы является разработка эффективного метода и инструмента идентификации веществ на основании их спектров.

Для её достижения решаются следующие задачи:

- нормализовать данные, сократить их размерность и влияние шумов за счёт реализации алгоритмов предобработки данных;
- реализовать нейронную сеть по типу многослойного персептрона для идентификации классифицируемых спектров;
- подобрать параметры сети, такие как количество слоёв, количество нейронов в скрытых слоях, вид функции активации и количество входных сигналов, так, чтобы *средняя энергия ошибки*, вычисляемая по формуле (3)

$$e_j(n) = d_j(n) - y_j(n) \quad (1)$$

$$E(n) = \frac{1}{2}\sum_{j \in C} e_j^2(n) \quad (2)$$

$$E_{av}(n) = \frac{1}{N}\sum_{n=1}^{N} E(n) \quad (3)$$

была минимальной на обучающем и валидационном множествах,

избежав тем самым переобучения сети. $E_{av}(n)$ представляет собой *функцию стоимости* (cost function) – меру эффективности обучения. В формулах (1) и (2) j – индекс нейрона в выходном слое, $e_j(n)$ – разность между выходом сети в этом нейроне ($y_j(n)$) и ожидаемым значением ($d_j(n)$), n – номер итерации, а N – количество сигналов в эпохе;

- проанализировать полученные результаты.

Обзор литературы

Прежде чем перейти непосредственно к описанию методов решения задачи, введём термины, которые будут использоваться в дальнейшем для описания проделанной работы.

Спектр – это последовательность квантов энергии электромагнитных колебаний, выделившихся, поглощённых или рассеявшихся при переходе атомов и молекул из одного энергетического состояния в другое. Спектроскопические методы анализа – это методы, основанные на взаимодействии материи и электромагнитного излучения. В данной работе рассмотрены три диапазона частот: ближний ультрафиолетовый, видимый и ближний инфракрасный [6].

Ультрафиолетовая спектроскопия – это раздел оптической спектроскопии, в который входит получение и изучение спектров в ультрафиолетовом диапазоне, то есть в диапазоне длин волн 10–400 нм [6].

Для получения ультрафиолетовых спектров используют дугу постоянного или переменного тока, пламя, низковольтные и высоковольтные искры, высокочастотные и сверхвысокочастотные разряды, плазмотроны, лазерное излучение. Зачастую при этом возникает проблема, что соответствующее оборудование является дорогим, потому для исследования ближнего УФ диапазона был использован светодиод. Приёмниками УФ излучения при длине волны более 230 нм служат обычные фотоматериалы. Поскольку при облучении в ультрафиолетовом диапазоне объект не изменяется и не разрушается, то это позволяет получить полные данные о его составе и строении. Вещества и объекты в ультрафиолетовом диапазоне часто обладают иными оптическими свойствами, нежели в видимой области. Можно наблюдать повышение коэффициента поглощения значительной части материальных объектов, прозрачных в видимой области. В качестве примера можно привести диэлектрики, такие как стекло, которые являются непрозрачным при длине волны менее 320 нм [6].

Инфракрасная спектроскопия изучает спектры в инфракрасной области, то есть в диапазоне длин волн от 780 нм до 1 мм. Инфракрасная область спектра разделяется на несколько диапазонов. Это разделение обусловлено применяемыми оптическими материалами, которые должны быть прозрачными в данном спектральном диапазоне:

- Ближняя инфракрасная область (0,8–2 мкм). Используется кварцевая и стеклянная оптика.
- Фундаментальная инфракрасная область (2–40 мкм). Исследуется при помощи солевой оптики или дифракционных решёток.
- Далёкая инфракрасная область (40–200 мкм). Исследуется при помощи дифракционных решёток.

Имеющееся оборудование для идентификации спектров в основном опирается на методы сравнения. Эти методы имеют достаточно низкую точность и не пригодны для работы с большой базой эталонов. Программа сравнивает полученный спектр с каждым имеющимся в базе эталоном и относит его к классу, в котором находится наиболее близкий по спектру объект [1]. Эти методы являются малоэффективными как по соображениям качества идентификации, так и по соображениям скорости их работы. Из-за необходимости сравнивать идентифицируемый спектр со всеми экземплярами, имеющимися в базе, данный метод работает очень долго, так как размеры сравниваемых векторов и количество экземпляров в базе зачастую превышают несколько тысяч. В итоге он далеко не всегда на практике даёт хорошие результаты.

Согласно имеющимся исследованиям о решении подобной задачи, спектры разных пород древесины имеют некоторые сходства и различия. Данные спектры можно различить по абсолютной величине или по первым производным спектральных функций. Авторами работы [7] был разработан алгоритм распознавания, принимающий во внимание данные показатели. Этот метод учитывает статистическую спектральную информацию об отдельных

породах, на основе чего строит так называемые «коридоры» минимальных и максимальных значений для указанных выше параметров. Таким образом, для всех спектров образцов одной породы строится коридор, в границах которого они находятся. Далее алгоритм для каждого нового образца ищет наиболее вероятное местоположение из всех таких коридоров [7].

Если провести анализ множества образцов древесины, можно заметить, что зачастую спектры образцов одной и той же породы могут отличаться друг от друга больше, чем от спектров другой породы. Понятие «коридора» в рамках одной породы не всегда можно сформировать, так как их может быть несколько. Кроме этого коридор одной породы может целиком содержаться в коридоре другой (см. Рис. 2).

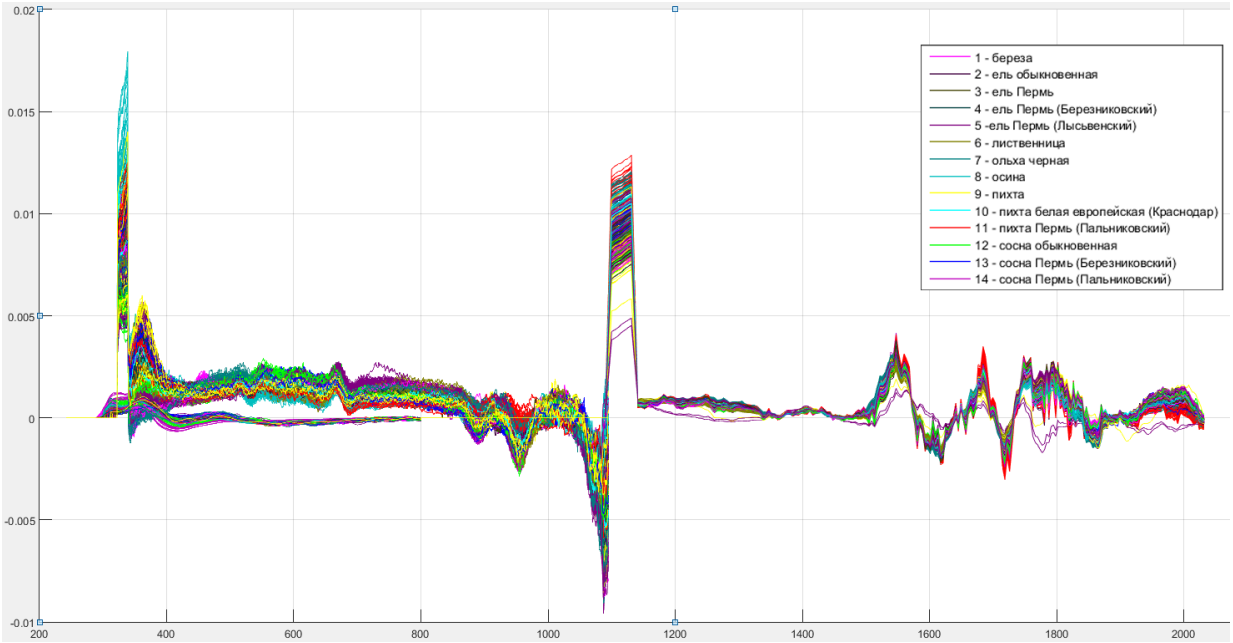


Рис. 3. Графики первых производных.

Если рассмотреть графики первых производных на Рис. 3, то можно заметить, что для некоторых пород древесины графики действительно местами образуют достаточно чётко наблюдаемые «коридоры», но они есть не для всех пород. Кроме вышесказанного вектор производных зависит от исходного вектора, а значит, он не несёт какой-то принципиально новой информации. Есть методы, часть из которых используется в данной работе, которые отслеживают взаимосвязи между

переменными эффективнее, чем дискретное взятие производной от вектора. Также в процессе взятия производной от вектора не сокращается размерность данных, в результате чего процедура идентификации при большом объёме данных может занимать неприемлемо большое время.

Глава 1. Предобработка данных

1.1. Метод главных компонент

Метод главных компонент (PCA) – это один из основных способов понижения размерности набора данных, состоящих из большого числа взаимосвязанных переменных, теряющий наименьшее количество информации касательно их взаимной вариации. Этот метод заключается в переносе пространства на новый ортогональный базис. Базисные векторы нового пространства направлены в сторону максимальной дисперсии набора входных данных. Дисперсия по направлению первого вектора нового базиса максимальна, вторая ось также максимизирует дисперсию, но добавляется условие ортогональности первому базисному вектору, и т.д., последний базисный вектор имеет наименьшую дисперсию из всех возможных. Это линейное многообразие является в каком-то смысле оптимальным среди всех других подпространств той же размерности. Оптимальность понимается в том смысле, что полученные проекции данных наилучшим образом отражают дисперсию исходных данных [8].

Данное отображение даёт возможность понизить количество информации, а точнее, количество признаков исходных данных через удаление осей координат, которые соответствуют базисным векторам с наименьшей дисперсией. Это делается из расчёта на то, что если необходимо отказаться от одной из осей, то лучше, если это будет та ось, по направлению которой набор входных данных меняется наименее существенно [9].

После определения, что такое главные компоненты, необходимо понять, как их найти. Пусть $X = \{x_1, \cdots, x_m\}$ – матрица исходных данных размера $[n \times m]$, где $x_{1 \cdots m}$ – вектора признаков для всех образцов, в данном случае спектров древесины, размера n (в данном случае $n = 604,\ m = 4352$). Классическая реализация метода главных компонент предполагает вычисление ковариационной матрицы:

$$cov(X) = C = [\mathrm{c}_{ij}]$$

$$\mathrm{c}_{ij} = \frac{1}{m-1}\sum_{k=1}^{m}(x_{ki} - \overline{x_i})(x_{kj} - \overline{x_j})$$

После чего необходимо найти такой вектор t_1 единичной длины, который будет максимизировать дисперсию, то есть $\max_{t_1^T t_1 = 1}(t_1^T C t_1)$. Для решения этой задачи можно воспользоваться стандартным методом множителей Лагранжа, максимизировав выражение

$$t_1^T C t_1 - \lambda(t_1^T t_1 - 1),$$

где λ – множитель Лагранжа. Дифференцируя по t_1 получим:

$$Ct_1 - \lambda t_1 = (C - \lambda E)t_1 = 0,$$

где E – единичная матрица размера $[m \times m]$. Таким образом, λ – собственное число C, а t_1 – собственный вектор. Для выбора конкретного вектора обратимся к выражению, которое максимизируется:

$$t_1^T C t_1 = t_1^T \lambda t_1 = \lambda t_1^T t_1 = \lambda,$$

следовательно, необходимо взять собственный вектор, который соответствует наибольшему собственному числу. Для выбора второй главной компоненты необходимо добавить условие, что она не должна коррелировать с первой, следовательно, она должна быть ей перпендикулярна $t_2^T t_1 = 0$. Получаем новое максимизируемое выражение

$$t_2^T C t_2 - \lambda_1(t_2^T t_2 - 1) - \lambda_2 t_2^T t_1,$$

где λ_1 и λ_2 – множители Лагранжа. Дифференцируя по t_2, получаем:

$$Ct_2 - \lambda_1 t_2 - \lambda_2 t_1 = 0,$$

а затем, умножив это равенство слева на t_1^T, получим:

$$t_1^T C t_2 - \lambda_1 t_1^T t_2 - \lambda_2 t_1^T t_1 = 0$$

$$0 - 0 - \lambda_2 \cdot 1 = 0$$

$$\lambda_2 = 0,$$

следовательно,

$$Ct_2 - \lambda_1 t_2 = (C - \lambda_1 E)t_2 = 0,$$

значит, λ_1 – ещё одно собственное число C, а t_2 – соответствующий ему собственный вектор. λ_1 также должно быть наибольшим, но не должно совпадать с предыдущим выбранным собственным числом, так как в противном случае $t_2 = t_1$, а это нарушит условие $t_2^T t_1 = 0$. Из этого следует, что λ_1 – второе по величине собственное число C, а t_2 – соответствующий ему собственный вектор. Для нахождения следующих главных компонент проводится аналогичная процедура. В результате получаем, что

$$[T\ \Lambda] = eig(C).$$

Основной формулой метода главных компонент является

$$X = TP^T,$$

где T – матрица *счетов* (scores), а P – матрица *нагрузок* (loadings). Матрица T состоит из столбцов t_i – собственных векторов матрицы C, отсортированных в порядке убывания (иногда в порядке возрастания) соответствующих им собственных чисел $\lambda_1 \geq \lambda_2 \geq \cdots \geq \lambda_n \geq 0$, которые находятся на диагонали матрицы Λ. Сокращение размерности происходит за счёт того, что оставляются только первые k векторов, соответствующие наибольшим собственным числам:

$$X = T_k P_k^T + o(X)$$

$$P_k = XT_k.$$

Часто метод главных компонент ассоциируют с сингулярным разложением матрицы (SVD). Сингулярное разложение матрицы – это разложение вида:

$$X = U\Sigma V^* \text{ или } X = U\Sigma V^T,$$

где Σ – прямоугольная диагональная матрица размера $m \times n$, на диагонали которой сингулярные числа, U (порядка m) и V (порядка n) – унитарные ($VV^* = E, UU^* = E$ или $VV^T = E, UU^T = E$) матрицы левых и правых сингулярных векторов соответственно:

$$T = U\Sigma,\ P = V.$$

Для достижения лучших результатов (за счёт уравнивания вклада всех

переменных) исходную матрицу X следует предварительно отцентрировать и нормировать (вычесть из каждого её элемента среднее значение по столбцу и разделить на стандартное отклонение по столбцу) [10].

Для рассматриваемых данных этот метод даёт очень хорошие результаты: общее количество признаков для каждого образца удалось сократить с $2048 + 2048 + 256 = 4352$ до 75 (количество главных компонент), сохранив при этом 99,9% информации. Полученный результат говорит о том, что мультиколлинеарность исходных данных очень высока. Выбор количества переменных был сделан из оценки относительного вклада собственных чисел в их сумму. Можно выбрать количество главных компонент другими способами, например, рассмотреть график величин собственных чисел и выбрать точку, начиная с которой величина собственных чисел практически не изменяется (см. Приложение 2. Графики величин собственных чисел). Визуально результат данного преобразования представлен на Рис. 4.

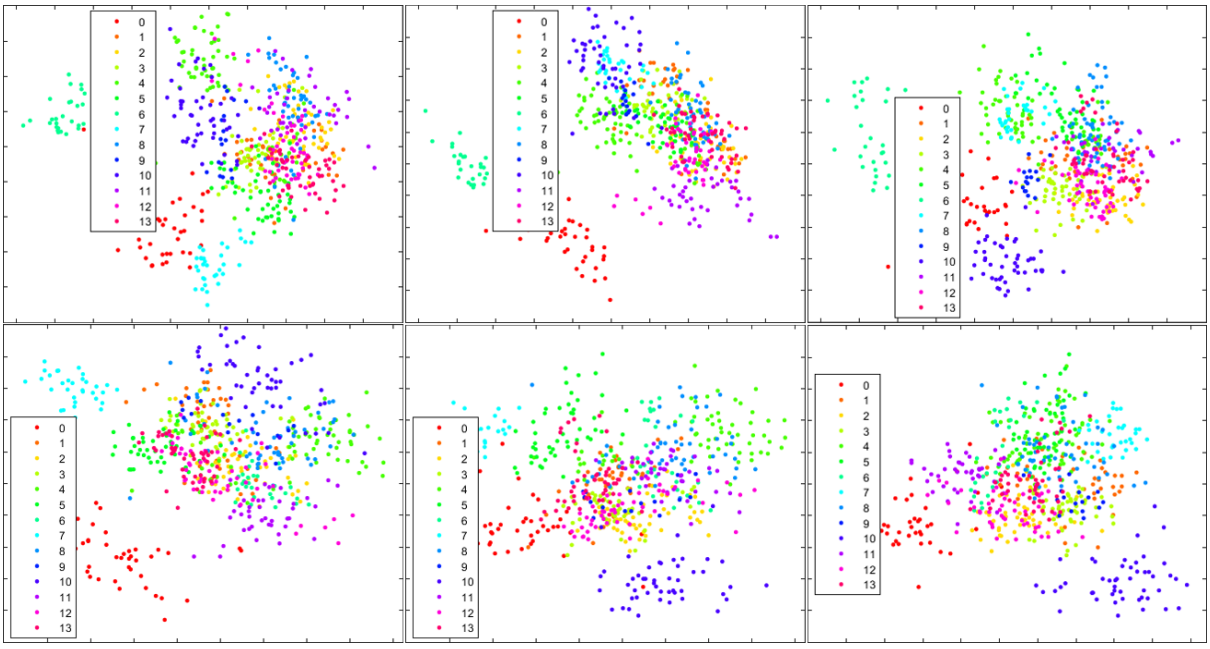


Рис. 4. Проекции на первые четыре главные компоненты после применения PCA.

При необходимости можно рассмотреть разницу между исходными данными и сжатыми, проделав обратное преобразование и рассмотрев разность (см. Рис. 5):

$$X_{new} = P_k T_k^T$$

$$\Delta X = X - X_{new}$$

Пример реализации в MATLAB:

```
[n, ~] = size(inputs);
mean_inputs = mean(inputs);
std_inputs=std(inputs);
autoscaling_inputs = (inputs - repmat(mean_inputs ,[n 1])) ./ repmat(std_inputs ,[n 1]);
[T, L] = eig(cov(autoscaling_inputs));
fractional_coverage_inputs = cumsum(diag(L)) / sum(diag(L))
T_reduced = T(:, (fractional_coverage_inputs > 0.001));
P = autoscaling_inputs * T_reduced;
```

Также можно использовать встроенную функцию `pca`.

```
inputs_recover = ((P*T_reduced').*repmat(std_inputs ,[n 1]))+repmat(mean_inputs ,[n 1]);
inputs_delta = inputs-inputs_recover;
```

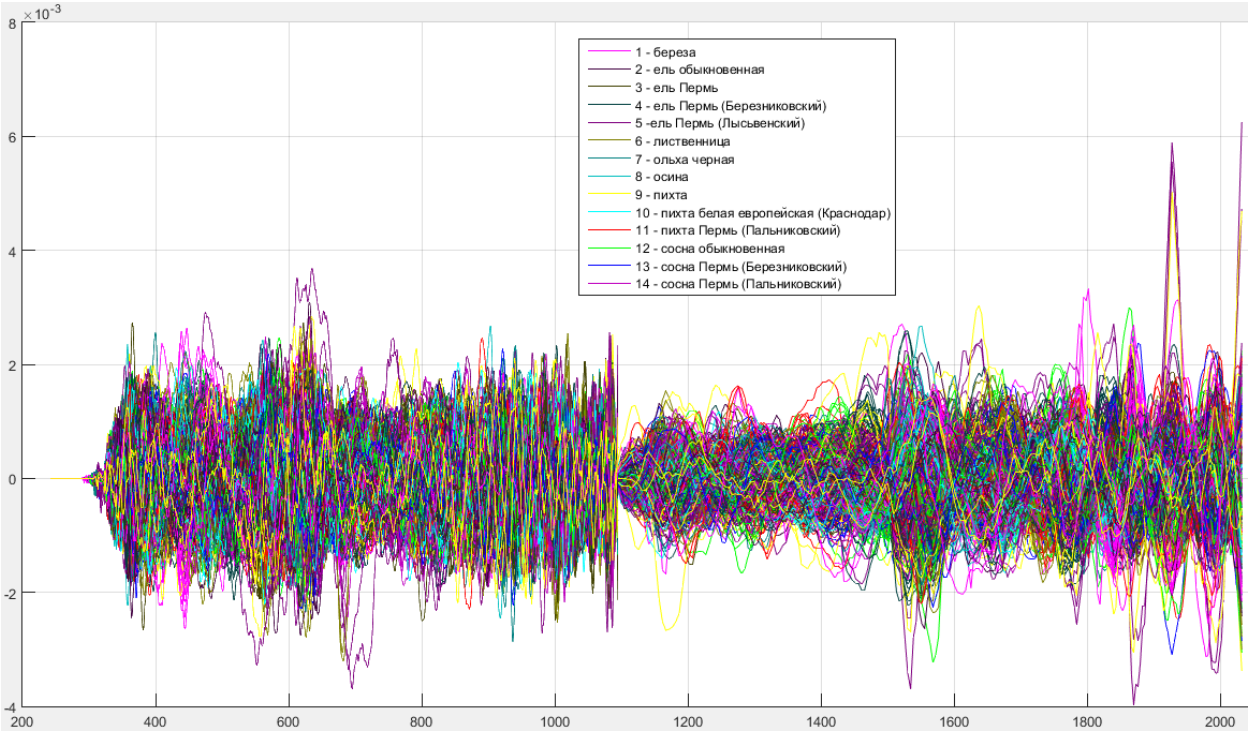


Рис. 5. График потерянной информации при преобразовании PCA.

Рассмотрев графики на Рис. 5, можно заметить, что по абсолютному значению они относительно малы и, по большей части, их поведение похоже на поведение шума.

Необходимо отметить, что метод главных компонент имеет в основе некоторые предположения:

- предположение о том, что с помощью линейного преобразования размерность данных может быть эффективно понижена;
- предположение о том, что направления, в которых дисперсия входных

данных максимальна, несут большее количество информации.

Несложно заметить, что данные предположения не всегда имеют место. Можно рассмотреть случай, когда все точки исходного множества находятся на поверхности гиперсферы, тогда линейное преобразование не сможет понизить размерность (однако, это сделает нелинейное преобразование, которое опирается на дистанцию от точки до центра сферы). Этот недостаток в равной степени присущ всем линейным алгоритмам и может быть решён с помощью использования вспомогательных переменных, которые будут являться нелинейными функциями от признаков набора исходных данных.

Второй недостаток метода главных компонент заключается в том, что направления, вдоль которых дисперсия максимальна, не всегда соответствуют максимальной информативности. Эта проблема связана с тем, что метод главных компонент не выполняет разделение классов, регрессию или прочие аналогичные действия, а лишь даёт возможность оптимальным образом восстановить исходный вектор из неполной информации о нем. Вся вспомогательная информация, связанная с вектором (например, что образ относится к какому-то классу), игнорируется. Поэтому во избежание потери компонент, содержащих информацию о вариации между классами, сокращение информации было столь незначительным – 0,1%. Данный метод был использован как подготовка данных для метода линейного дискриминантного анализа, который учитывает принадлежность образа к классу и для которого является важным отсутствие корреляции между переменными (признаками).

1.2. Линейный дискриминантный анализ

Линейный дискриминантный анализ (LDA) – это метод статистики и машинного обучения, который используется для нахождения линейной комбинации признаков, наилучшим образом разделяющей два или более класса объектов. Линейный дискриминантный анализ осуществляет проекцию пространства векторов признаков R^n таким образом, чтобы минимизировать

внутриклассовое и максимизировать межклассовое расстояния в новом пространстве признаков R^m, которое является подпространством исходного и, как правило, имеет меньший размер ($m \leq n$). Целю метода является предельное упрощение процесса классификации, а именно извлечение пространства, в котором спроецированные исходные вектора признаков, относящиеся к разным классам, максимально удалены друг от друга. Пока центроиды классов располагаются достаточно далеко друг от друга, перекрытие границ классов не является критичным [11].

Формально можно описать задачу и её решение следующим образом:

- имеется k классов – $\{\omega_1, \omega_2, \dots, \omega_k\}$
- средний вектор для класса ω_i

$$\overline{x_\iota} = \frac{1}{N_i} \sum_{j=1}^{N_i} x_{i,j}$$

  N_i – количество объектов в классе ω_i, $x_{i,j}$ – j-й объект класса ω_i (вектор)

- матрица ковариаций для класса ω_i

$$\Sigma_i = \frac{1}{N_i - 1} \sum_{j=1}^{N_i} (x_{i,j} - \overline{x_\iota})(x_{i,j} - \overline{x_\iota})^T$$

- средний вектор для всей выборки

$$\bar{x} = \frac{1}{N} \sum_{i=1}^{k} \sum_{j=1}^{N_i} x_{i,j} = \frac{1}{N} \sum_{i=1}^{k} N_i \, \overline{x_\iota}$$

- внутриклассовая матрица рассеяния (Within class scatter matrix)

$$S_w = \sum_{i=1}^{k} (N_i - 1)\Sigma_i = \sum_{i=1}^{k} \sum_{j=1}^{N_i} (x_{i,j} - \overline{x_\iota})(x_{i,j} - \overline{x_\iota})^T$$

- межклассовая матрица рассеяния (Between class scatter matrix)

$$S_b = \sum_{i=1}^{k} N_i (\overline{x_\iota} - \bar{x})(\overline{x_\iota} - \bar{x})^T$$

$\bar{x_i}$ – средний вектор класса ω_i, $\bar{x}$ – общий средний вектор.

Цель:

$$\Phi_{LDA} = \arg\max_{\Phi} \frac{|\Phi^T S_b \Phi|}{|\Phi^T S_w \Phi|} \quad (4)$$

- найти проецирующую матрицу Φ_{LDA}, которая максимизирует соотношение (4);
- поскольку смысл определителя ковариационной матрицы есть величина дисперсии, то разыскивается проекция, максимизирующая дисперсию средних элементов классов и минимизирующая дисперсию внутри самих классов.

Решение:

- Φ_{LDA} – это решение уравнения:

$$S_b \Phi - S_w \Phi \Lambda = 0$$

- Умножим на S_w^{-1}:

$$S_w^{-1} S_b \Phi - S_w^{-1} S_w \Phi \Lambda = 0$$

$$S_w^{-1} S_b \Phi - \Phi \Lambda = 0$$

$$S_w^{-1} S_b \Phi = \Phi \Lambda$$

следовательно, Φ_{LDA} состоит из собственных векторов $S_w^{-1} S_b$ при условии, что матрица S_w^{-1} существует. Её существование обусловлено тем, что предварительно был применён метод главных компонент, который избавил данные от мультиколлинеарности [12].

Φ_{LDA} получается из тренировочных данных, поэтому для завершения работы остаётся только перевести все данные, включая тестовые, в новое пространство:

$$x_{new} = (x^T \Phi_{LDA})^T.$$

Аналогично методу главных компонент в данном методе можно осуществить сокращение размерности, оставив только те собственные векторы $S_w^{-1} S_b$, которые соответствуют наибольшим собственным числам. Визуально результат данного преобразования можно увидеть на Рис. 6.

В дальнейшем при подборе оптимальных параметров изменяется только количество главных компонент после применения LDA, так как PCA был использован исключительно с целью подготовки данных для LDA.

В связи с тем, что в MATLAB нет встроенной поддержки межклассового (количество классов > 2) линейного дискриминантного анализа, реализация данного метода достаточно объёмна (см. Приложение 1. Реализация LDA в MATLAB).

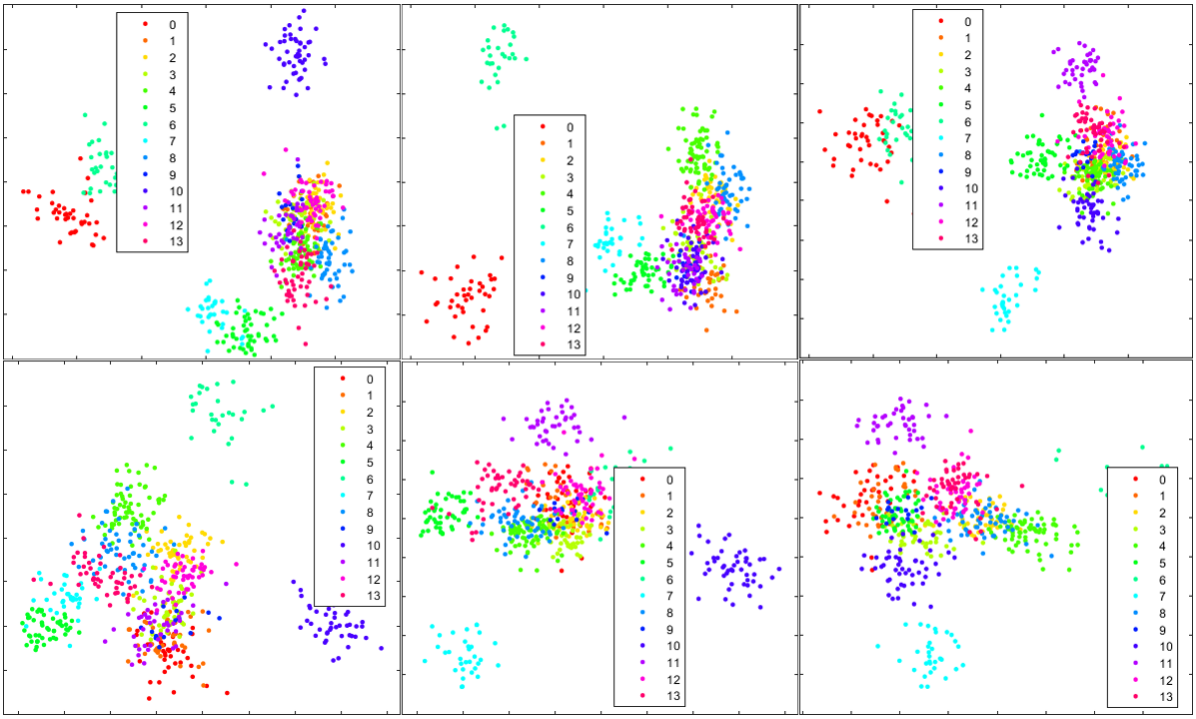


Рис. 6. Проекции на первые четыре главные компоненты PCA+LDA.

Несмотря на то, что LDA работает с информацией о принадлежности объекта к одному из классов, он не является алгоритмом классификации. Одной из целей применения метода является уменьшение количества признаков исходных данных для дальнейшего применения нелинейного алгоритма классификации. В качестве такого классификатора была использована нейронная сеть по типу многослойного персептрона с нелинейной функцией активации.

Глава 2. Нейронная сеть

2.1. Описание нейронной сети

Нейронная сеть устроена по типу многослойного персептрона и используется в качестве классификатора. Сети такого типа успешно используются для решения широкого круга различных сложных задач прогнозирования или распознавания.

Одним из общепринятых алгоритмов обучения многослойного персептрона *с учителем* является алгоритм *обратного распространения ошибки* (error back-propagation algorithm) [3], который основывается на коррекции ошибок отклика сети (error correction learning rule), потому его можно рассматривать как обобщение столь же популярного алгоритма минимизации среднеквадратической ошибки (LMS) [3].

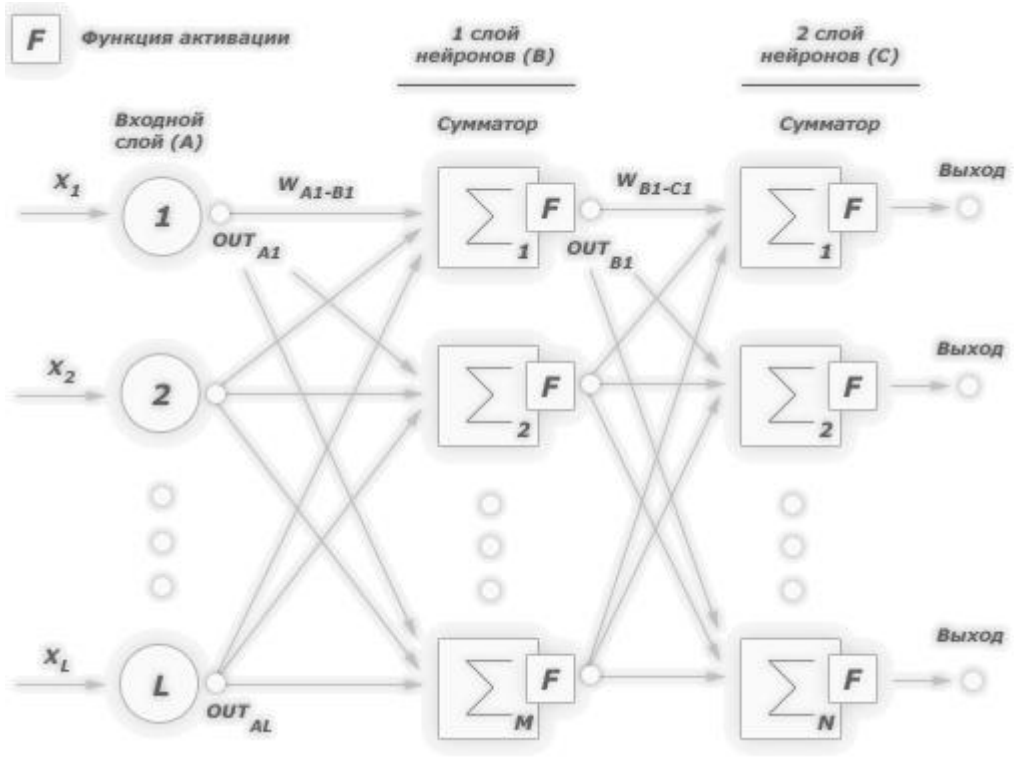


Рис. 7. Схема нейронной сети с одним скрытым слоем.

Для решения задачи идентификации спектров после проделанный предобработки данных оказалось достаточно одного скрытого слоя, таким образом, функционально важных слоёв два, при учёте выходного слоя

(см. Рис. 7). Сеть является полносвязной. Сигнал передаётся по сети в прямом направлении, от входного слоя к выходному слою, следующим образом:

- начиная с первого скрытого слоя, для каждого нейрона рассчитывается *индуцированное локальное поле*

$$v_j^l(n) = \sum_{i=1}^{m_{l-1}} w_i^j(n) \cdot y_i^{l-1}(n) + b_j^l,$$

где l – номер текущего слоя, j – номер нейрона в текущем слое, m_{l-1} – количество связей с предыдущим слоем, w_i^j – веса связей, по которым идёт входной сигнал, b_j^l – порог (bias), применяемый к нейрону, который можно представить в качестве синаптического веса $b_j^l = w_0^j$, соответствующего фиксированному входу $y_0^{l-1} = 1$;

- далее $v_j^l(n)$ подаётся на вход функции активации, в результате чего создаётся выходной сигнал нейрона

$$y_j^l(n) = \varphi\left(v_j^l(n)\right),$$

который будет передан на следующий слой, либо, в случае если текущий слой – выходной, войдёт в результат отклика сети.

В качестве функции активации была выбрана логистическая:

$$\varphi(x) = \frac{1}{1+e^{-ax}}, \quad \varphi'(x) = \frac{ae^{-ax}}{(1+e^{-ax})^2} = a\varphi(x)(1-\varphi(x)). \qquad (5)$$

Её область значений совпадает со значениями выходного слоя, она имеет сигмоидальный вид и легко вычисляемую производную, что необходимо для метода обратного распространения ошибки [13], с помощью которого обучается сеть. Нелинейность функции активации позволяет данному классификатору успешно справиться с задачей, когда множества классов не являются линейно сепарабельными. Если говорить о том, что в некоторых классах могут образовываться отдельные подгруппы, то, благодаря своей архитектуре, они не составят особой проблемы для данного классификатора, поскольку её можно решить при помощи увеличения количества нейронов в скрытом слое.

2.2. Обучение нейронной сети

Целью процесса обучения является настройка свободных параметров сети для минимизации величины $E_{av}(n)$ (3). Текущая энергия ошибки $E(n)$ (2), а, значит, и средняя энергия ошибки $E_{av}(n)$ являются функциями всех свободных параметров (т.е. синаптических весов и значений порога) сети.

Обучение начинается с инициализации весов нейронов случайным образом из промежутка $(0,1)$. При этом используется последовательный режим обучения, при котором элементы обучающего множества подаются на вход сети в перемешанном порядке для каждой эпохи. После каждого полученного ответа сети вычисляется энергия ошибки. Далее происходит минимизация величин $E_{av}(n)$ и $E(n)$ через корректировку весов с помощью метода обратного распространения ошибки. Веса обновляются для каждого обучающего примера в пределах одной эпохи (т.е. подачи на вход всего обучающего множества).

Предполагается, что при обучении методом обратного распространения ошибки происходит два прохода по всем слоям сети: *прямой* и *обратный*. В течение прямого прохода (forward pass) на сенсорные узлы подаётся образ (входной вектор), который далее распространяется от слоя к слою по всей сети, при этом все синаптические веса сети фиксированы. В результате прямого прохода генерируется набор выходных сигналов (вектор), который представляет собой реакцию сети на данный входной вектор. В процессе обратного прохода (backward pass) происходит настройка синаптических весов по правилу коррекции ошибок: из желаемого (целевого) отклика вычитается фактический выход сети, в результате чего формируется сигнал ошибки (error signal) (1). После этого он распространяется по сети справа налево от слоя к слою с параллельным вычислением *локального градиента* (7) для каждого нейрона. Именно поэтому алгоритм и получил название обратного распространения ошибки.

Локальный градиент нейронов, расположенных в выходном слое, получается из соответствующего сигнала ошибки, умноженного на производную нелинейной, в данном случае логистической, функции активации. На основе вычисления локальных градиентов для всех нейронов выходного слоя вычисляются локальные градиенты для всех нейронов предыдущего слоя, из которых можно получить величину коррекции связей. Аналогичные вычисления проводятся для всех слоёв в обратном направлении:

$$\Delta w_i^j(n) = -\eta \frac{\partial E(n)}{\partial w_i^j(n)} = -\eta \frac{\partial E(n)}{\partial e_j(n)} \frac{\partial e_j(n)}{\partial y_i^l(n)} \frac{\partial y_i^l(n)}{\partial v_j^l(n)} \tag{6}$$

$$\Delta w_i^j(n) = \eta e_j(n) \varphi' \left(v_j^l(n)\right) y_i^{l-1}(n)$$

$$\delta_j^l(n) = e_j^l(n) \varphi' \left(v_j^l(n)\right) = \varphi' \left(v_j^l(n)\right) \sum_{i=1}^{m_{l+1}} \delta_j^{l+1}(n) w_j^i(n) \tag{7}$$

$$\Delta w_i^j(n) = \eta \delta_j^l(n) y_i^{l-1}(n). \tag{8}$$

В равенстве (8) локальный градиент $\delta_j^l(n)$ из формулы (7) указывает на требуемое изменение синаптического веса. Знак выражения (6) обусловлен направлением антиградиента, так как производится процедура градиентного спуска по поверхности ошибок. В формулах (6), (7) и (8) η – *параметр скорости обучения* – константа, как правило, из промежутка $(0,1]$. Это обусловлено тем, что если сильно увеличить параметр η с целью увеличения скорости обучения, то это может привести систему в неустойчивое состояние. Но можно повысить скорость обучения и без потери устойчивости, добавив к $\Delta w_i^j(n)$ *момент инерции* [13]:

$$\Delta w_i^j(n) = \alpha \Delta w_i^j(n-1) + \eta \delta_j^l(n) y_i^{l-1}(n), \qquad \alpha \in [0,1).$$

Для того, чтобы вычислить локальный градиент по формуле (7) необходимо знать значение производной сигмоидальной функции активации φ – следовательно, она должна быть непрерывно дифференцируемой. В качестве такой функции в многослойных персептронах часто используется логистическая функция (5) или функция гиперболического тангенса. Однако,

гиперболический тангенс имеет область значений $[-1,1]$, в результате чего эта функция менее пригодна для данной задачи, так как ожидаемый выходной вектор предположительно состоит из 0 и 1. Выходной вектор имеет размер количества классов (в данной задаче классов 14), и единица должна стоять в том элементе вектора, индекс которого соответствует номеру класса (к которому относится образец).

Существует множество критериев остановки обучения. В данной задаче лучше всего себя показал критерий, основывающийся на достижении минимума средней энергии ошибки E_{av} на валидационном множестве. При его достижении текущие веса сохраняются как оптимальные, и, в случае если за определённое количество эпох не удалось уменьшить значение E_{av} на валидационном множестве, то сеть восстанавливает сохранённые значения весов и заканчивает обучение, после чего она готова к подаче на вход тестовых образцов и их идентификации.

Прилагается реализация структуры сети и её процесса обучения (см. Приложение 4. Реализация структуры нейронной сети и её процесса обучения.). Для реализации нейронной сети в MATLAB можно использовать `nntool` или `nnstart` (Neural Network Toolbox).

2.3. Перекрёстная проверка

Во избежание переобучения сети, а также для объективной оценки работы данного стека методов используется метод *перекрёстной проверки* или *кроссвалидации* (cross validation) [14]. Разделение на обучающее, валидационное и тестовое множества происходит следующим образом:

- все сигналы случайным образом распределяются на c примерно равных групп. c выбрано равным 10;
- получается c итераций, в которых сеть каждый раз заново обучается на $\frac{c-2}{c}$ от общего числа сигналов, проходит валидацию на $\frac{1}{c}$ с целью сохранить аппроксимирующие свойства (хорошую обобщающую

способность) и тестируется на $\frac{1}{c}$ (от общего количества сигналов).

Функция разбиения на группы на языке C++:

```
//функция генерации случайных индексов обучающей, валидационной и тестовой
   выборок
void clustering(const int number_of_groups)
{
    crossvalidate_groups.resize(number_of_groups);
    for (int i = 0; i < x.size(); i++)
    {
        int ran = rand() % number_of_groups;
        crossvalidate_groups[ran].push_back(i);
    }
}
```

Организация самого процесса кроссвалидации на языке C++ достаточно объёмна (см. Приложение 3. Реализация процесса кроссвалидации). Для организации процесса на MATLAB можно использовать встроенные методы `crossval`.

Глава 3. Оценка эффективности применения PCA, LDA и нейронной сети для обработки и идентификации спектров

3.1. Выбор количества главных компонент

Выбор количества главных компонент после применения LDA осуществляется путём анализа графика средней точности идентификации тестового множества для всех итераций кроссвалидации. Конфигурация нейронной сети фиксируется на заведомо избыточном количестве нейронов скрытого слоя – на 20 нейронах.

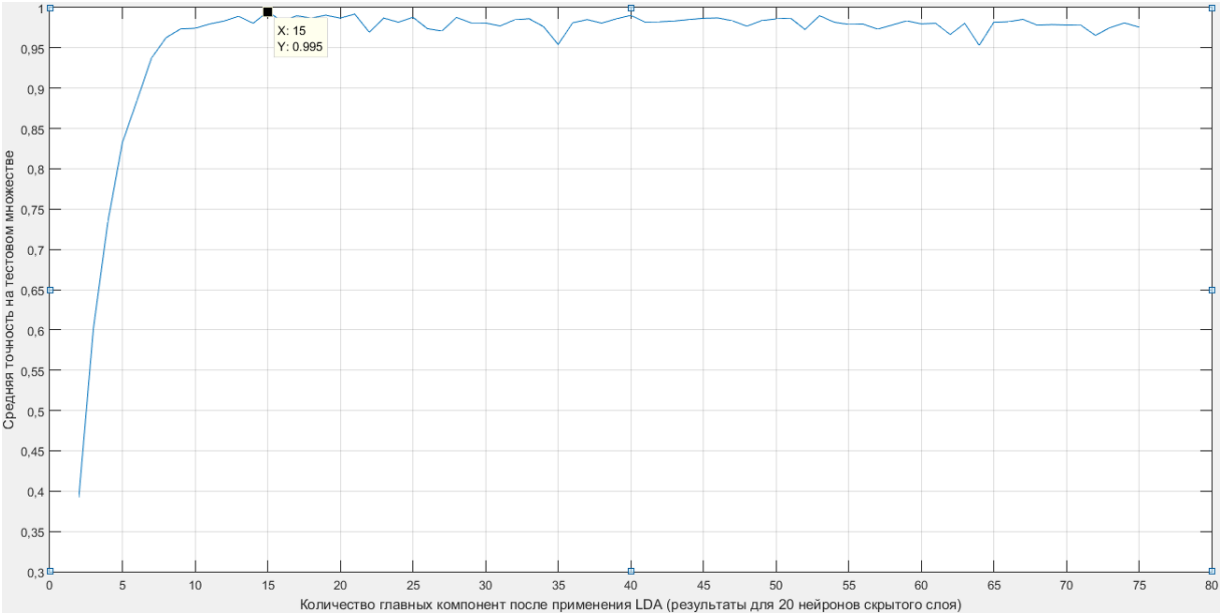


Рис. 8. График зависимости средней точности от количества входных сигналов.

Из графика (см. Рис. 8) видно, что 15 главных компонент достаточно для достижения более чем приемлемой средней точности на тестовом множестве – 99,5%. Брать большее количество главных компонент нецелесообразно, так как это отразится на быстродействии, но не даст существенного прироста точности.

3.2. Выбор количества нейронов скрытого слоя

Для выбора количества нейронов скрытого слоя выполняется процедура, аналогичная выбору количества главных компонент. Фиксируется количество входных сигналов и изменяется количество нейронов. На предыдущем шаге было получено, что оптимальное количество главных компонент (признаков) – входных сигналов нейронной сети – равно 15.

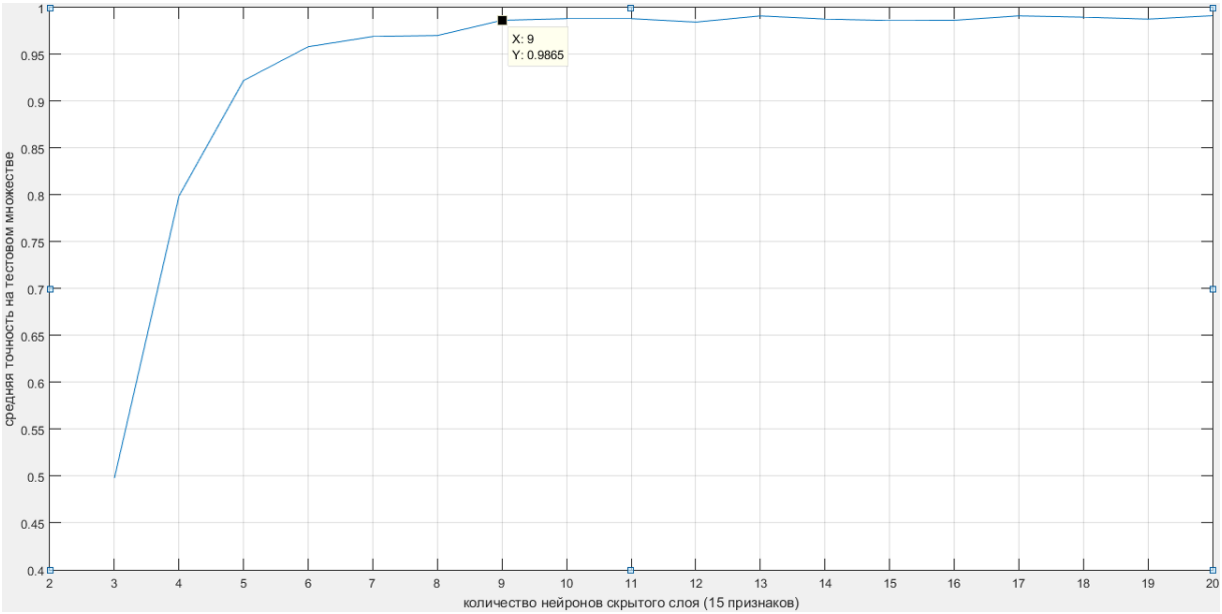


Рис. 9. График зависимости средней точности от количества нейронов скрытого слоя.

Из графика (см. Рис. 9) видно, что 9 нейронов скрытого слоя достаточно для достижения 98,65% средней точности на тестовом множестве.

3.3. Оценка результатов

На идентификацию одного тестового экземпляра уходит 0,0003 секунды. Из этого следует, что за секунду можно идентифицировать более 3300 экземпляров, при этом размер базы экземпляров не влияет на скорость идентификации. 98,65% средней точности на тестовом множестве – очень хороший результат. Для сравнения, стандартные методы классификации, такие как *k ближайших соседей* (k-nearest neighbors) или *k ближайших взвешенных соседей*, верно идентифицируют не более 65%, а лучший средний результат при процедуре кроссвалидации – 44,3%.

Многослойный персептрон хорошо подходит для задачи классификации веществ по их спектрам (см. график минимизируемой функции E_{av} на Рис. 10), в особенности, если выполнить предварительную обработку данных. Алгоритм относительно прост в реализации и позволяет эффективно решать сложные задачи.

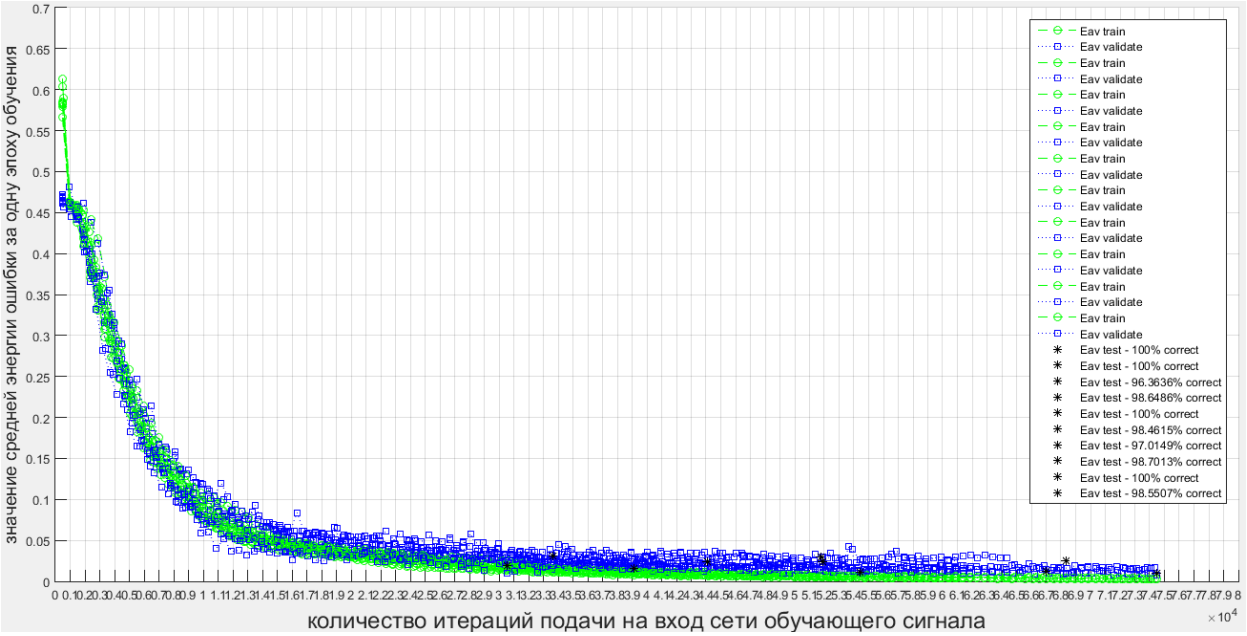


Рис. 10. График изменения средней энергии ошибки в процессе обучения нейронной сети

Выводы

В ходе выполнения работы был разработан эффективный метод и инструмент идентификации веществ на основании их спектров. Были решены следующие задачи:

- произведена качественная предобработка данных;
- выбран подходящий метод классификации;
- подобраны оптимальные параметры;
- проведён объективный анализ результатов.

Исходя из того, что данный стек методов не использует конкретные особенности спектров древесины, а выявляет характерные признаки автоматически, можно сделать предположение, что он будет также эффективен при классификации других материалов со сложным составом и может быть использован, например, в производственных целях контроля качества или при проверке соответствия материала его накладным (например, при таможенном контроле).

Заключение

Полученный набор методов является универсальным для задачи классификации и не ограничивается идентификацией породы древесины по спектрам образцов. В целом, он может быть применён не только для задач классификации, так как нейронная сеть по типу многослойного персептрона имеет широкий круг потенциального использования, а PCA и LDA также достаточно универсальные методы.

Методы, описанные в данной работе, изначально были опробованы и протестированы в среде MATLAB, так как эта среда имеет большое количество встроенных функций и широкие возможности визуализации результатов. Когда подходящий стек методов для решения задачи был найден, код был переписан на C++, что сделало его бесплатным. Это было сделано для того, чтобы использовать данное решение на компьютерах, на которых нет MATLAB, принимая во внимание, что стандартная лицензия на MATLAB очень дорога: 2650$, что равно 170483₽ за одну копию.

Список литературы

1. Колгин Е. А., Ухов А. А., Савушкин А. В. Спектрометрическое устройство для идентификации пород древесины // Петербургский журнал электроники №2(55)–3(56). 2008. 127 с.
2. Зайдель А. Н. Основы спектрального анализа. // «Наука». 1965. 322 с.
3. Haykin S. Neural Networks: A Comprehensive Foundation, 2nd edition. Hamilton, Ontario: Pearson Education. 1999. 823 p.
4. Jolliffe I. T. Principal Component Analysis. New York: Springer. 2002. 518 p.
5. Rao R. C. The utilization of multiple measurements in problems of biological classification // Journal of the Royal Statistical Society. Series B (Methodological) / Published by: Wiley for the Royal Statistical Society. 1948. Vol. 10, No 2. P. 159–203.
6. Нагибина И. М., Прокофьев В. К. Спектральные приборы и техника спектроскопии. // Учебное пособие. 1967. 327 с.
7. Воронин А. А., Смирнова Е. В., Смирнов А. П. К вопросу идентификации пород древесины с применением методов анализа спектров. // Научно-технический вестник СПБГУ ИТМО №2(66). 2010. 5–11 с.
8. ALGLIB, «Метод главных компонент» http://alglib.sources.ru/dataanalysis/principalcomponentsanalysis.php.
9. LeCun Y. A. Efficient learning and second-order methods // A Tutorial at NIPS 93. Denver. 1993. 71 p.
10. Зиновьев А. Ю. Визуализация многомерных данных.

Красноярск: Издательство Красноярского государственного технического университета. 2000. 180 с.

11. ALGLIB, «Линейный дискриминантный анализ» http://alglib.sources.ru/dataanalysis/lineardiscriminantanalysis.php.
12. Duda R. O., Hart P. E., Stork D. G. Pattern classification. Toronto: Wiley. 2001. 738 p.
13. Rumelhart D. E., Hinton G. E., Williams R. J. Learning representations of backpropagation errors // Nature (London). 1986. Vol. 323. P. 533–536.
14. Amari S., Murata N., Muller K. R., Finke M., Yang H. Statistical theory of overtraining — is cross-validation asymptotically effective // Advances in Neural Information Processing Systems / Cambridge, MA: MIT Press. 1996. Vol. 8. P. 176–182.

Приложения

Приложение 1. Реализация LDA в MATLAB

LDA.m:

```
classdef LDA < handle
    %inherit from handle otherwise we must store value (object) each time when we change
    some propertie value
    %function would need to return obj(this)

    properties
        Samples, %array of feature vectors
        Classes, %class labels for samples

        EigenVectors,
        EigenValues,

        BetweenScatter, %SB
        WithinScatter,  %SW

        NumberOfClasses
    end

    properties(Access = 'private')
        ClassSubsetIndexes %start and end indicies where one class begins and ends in
        variable "Classes"

        TotalMean,
        MeanPerClass
    end

    methods

        function this = LDA(samples, classes)
            this.Samples = samples;
            this.Classes = classes;

            this.ClassSubsetIndexes = LDA.GetClassSubsetIndexes(classes);
            this.NumberOfClasses = size(this.ClassSubsetIndexes, 1);

            [this.TotalMean this.MeanPerClass] = this.CalculateMean(samples);
        end

        function Compute(this)
            SB = this.CalculateBetweenScatter();
            SW = this.CalculateWithinScatter();

            [eigVectors, eigValues] = eig(SB, SW);
            eigValues = diag(eigValues);

            %--------- sort eig values and vectors ---------%
            sortedValues = sort(eigValues,'descend');
            [c, ind] = sort(eigValues,'descend'); %store indicies
            sortedVectors = eigVectors(:,ind); % reorder columns

            this.EigenVectors = sortedVectors;
            this.EigenValues = sortedValues;

            this.BetweenScatter = SB;
            this.WithinScatter = SW;
        end
```

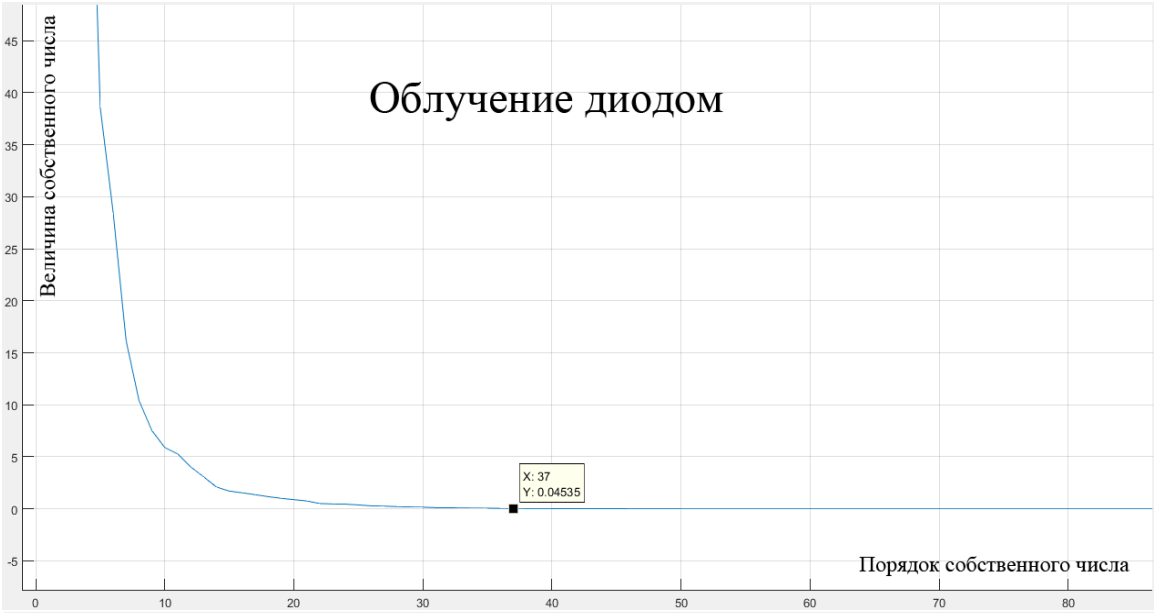
```
    function projectedSamples = Transform(this, samples, numOfDiscriminants)
        vectors = this.EigenVectors(:, 1:numOfDiscriminants);

        %transformed sample is scalar (projection on a hyperplane)
        projectedSamples = samples * vectors;
    end

    function measure = CalculateFLDMeasure(this, numOfDiscriminants)
        SB = this.BetweenScatter;
        SW = this.WithinScatter;

        vectors = this.EigenVectors(:, 1:numOfDiscriminants);

        measure = det(vectors' * SB * vectors) / det(vectors' * SW * vectors);
    end

end

methods(Access = 'private')
    function [totalMean meanPerClass] = CalculateMean(this, samples)

        for classIdx=1 : 1 : length(this.ClassSubsetIndexes)
            startIdx = this.ClassSubsetIndexes(classIdx, 1);
            endIdx = this.ClassSubsetIndexes(classIdx, 2);
            meanPerClass(classIdx, :) = mean( samples(startIdx:endIdx, :), 1);
        end

        totalMean = mean(meanPerClass, 1); %global average value
    end

    function SW = CalculateWithinScatter(this)
        featureLength = size(this.Samples, 2);
        SW = zeros(featureLength, featureLength);

        for classIdx=1 : 1 : length(this.ClassSubsetIndexes)
            startIdx = this.ClassSubsetIndexes(classIdx, 1);
            endIdx = this.ClassSubsetIndexes(classIdx, 2);

            classSamples = this.Samples(startIdx:endIdx, :);
            classMean = this.MeanPerClass(classIdx, :);
            Sw_Class = LDA.CalculateScatterMatrix(classSamples, classMean);

            SW = SW + Sw_Class;
        end
    end

    function SB = CalculateBetweenScatter(this)
        featureLength = size(this.Samples, 2);
        SB = zeros(featureLength, featureLength);

         for classIdx=1 : 1 : length(this.ClassSubsetIndexes)

            startIdx = this.ClassSubsetIndexes(classIdx, 1);
            endIdx = this.ClassSubsetIndexes(classIdx, 2);
            numberOfSamplesInClass = endIdx - startIdx + 1;

            classMean = this.MeanPerClass(classIdx, :);

            %because my vector is row-vector
            Sb_class = (classMean - this.TotalMean)' * (classMean - this.TotalMean);
            Sb_class = numberOfSamplesInClass * Sb_class;

            SB = SB + Sb_class;
         end

    end

end
```

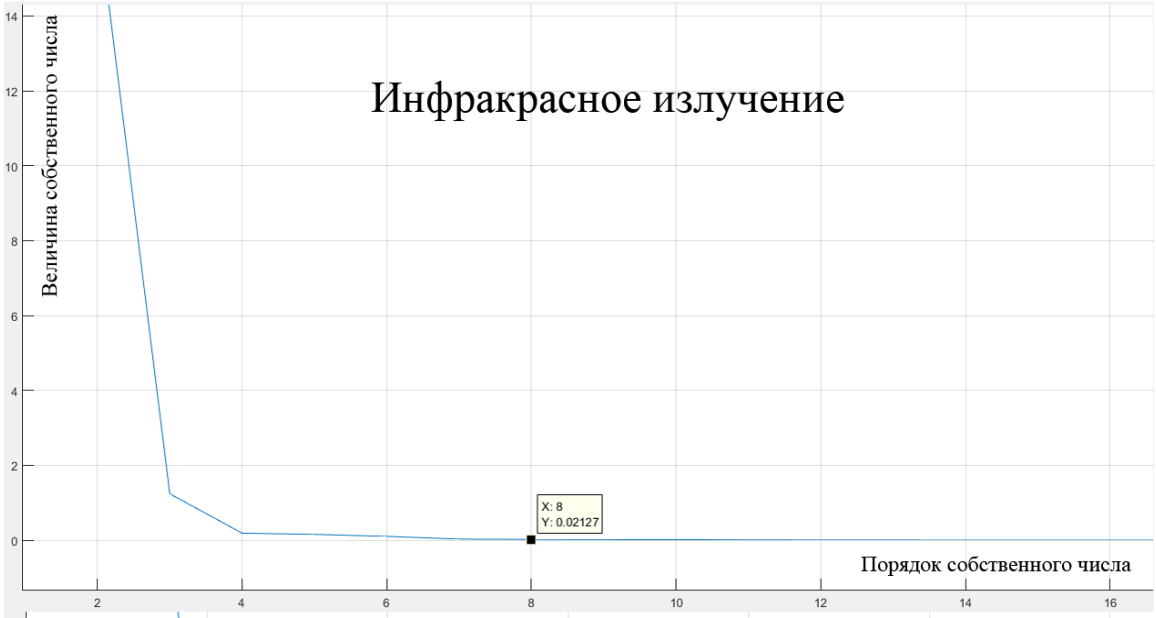
```
    methods(Static, Access = 'private')

        function subset = GetClassSubsetIndexes(classes)

            subset=[];
            oldClassLabel = 'nekaLabela';

            for i=1 : 1 : length(classes)
                if oldClassLabel ~= classes{i}
                    oldClassLabel = classes{i};

                    subset = cat(1, subset, i);
                end
            end

            %now put end indicies
            for i=2 : 1 : size(subset,1)
                endIndex = subset(i, 1);
                subset(i-1, 2) = endIndex-1;
            end
            subset(size(subset,1), 2) = length(classes);
        end

        function Sw_class = CalculateScatterMatrix(classSamples, classMean)

            featureLength = size(classSamples, 2);
            Sw_class = zeros(featureLength, featureLength);

            for sampleIdx=1 : 1 : size(classSamples, 1)
                covariance = (classSamples(sampleIdx, :) - classMean);
                covariance = covariance' * covariance; %because my vector is row-vector

                Sw_class = Sw_class + covariance;
            end
        end

    end

end
```

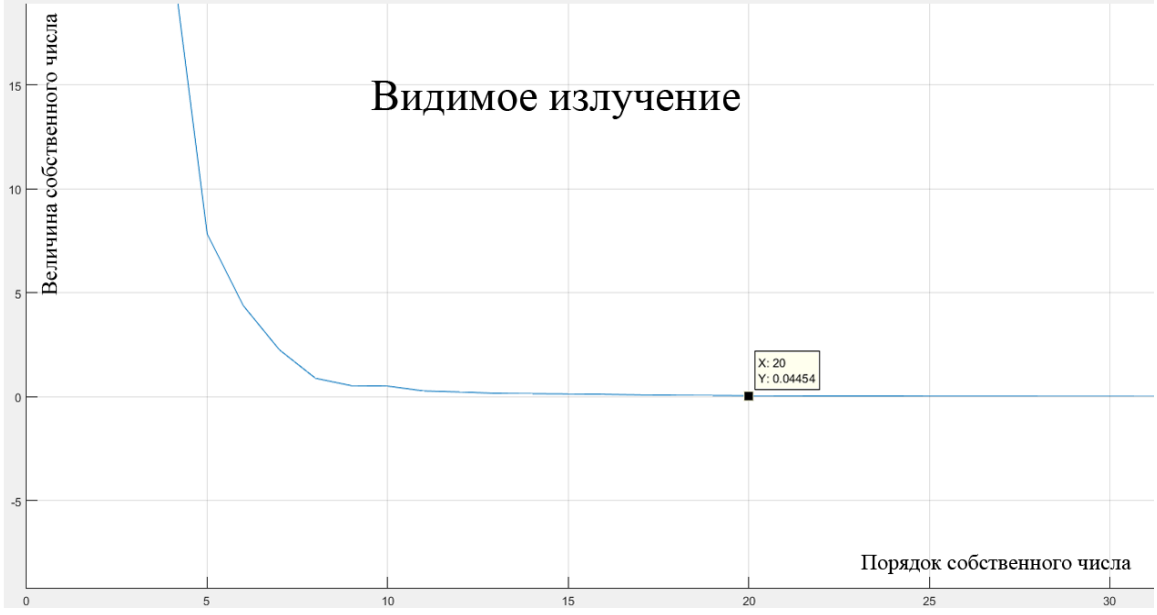
useLDA.m:

```
trainSamples = PCA_train_reuced;
testSamples = PCA_test_reuced;
trainClasses = targetstype2;

%************************ MultiClass LDA ************************************

mLDA = LDA(trainSamples, trainClasses);
mLDA.Compute();

LDA_PCA_train_reuced = mLDA.Transform(trainSamples, 15);
LDA_PCA_test_reuced = mLDA.Transform(testSamples, 15);

%************************ MultiClass LDA ************************************
```

Приложение 2. Графики величин собственных чисел







Приложение 3. Реализация процесса кроссвалидации

```
534     //возвращает среднюю точность на тестовом множестве
535     double cross_validation(bool logging)
536     {
537         ofstream out_train("logs//training_results.txt");
538         ofstream out_test_results("logs//testing_results.txt");
539         ofstream out_plot("MATLAB//plots\plot_E.m");
540         vector<vector<double> >old_x = x;
541         double results=0;
542         for (int i = 0; i < crossvalidate_groups.size(); i++)
543         {
544             int t = i, v = (t + 1) % crossvalidate_groups.size();
545             test = crossvalidate_groups[t];
546             validate = crossvalidate_groups[v];
547             for (int j = 0; j < crossvalidate_groups.size(); j++)
548             {
549                 if (j != t && j != v)
550                 {
551                     train.insert(train.end(), crossvalidate_groups[j].begin(),
                          crossvalidate_groups[j].end());
552                 }
553             }
554             PCA(x, train, test, validate);
555             LDA(x, targets,train, test, validate);
556             if (logging)
557             {
558                 out_train << "tain size = " << train.size() << "\ttest size =
                      " << test.size() << "\tvalidate size = " << validate.size()
                      << endl;
559                 sequential_training_mode(out_train);
560                 cout << "TESTING" << endl;
561                 testing(test, E_test_av, test_answers, out_test_results);
562                 plot_E(out_plot, i);
563             }
564             else
565             {
566                 cout << "tain size = " << train.size() << "\ttest size = " <<
                      test.size() << "\tvalidate size = " << validate.size() <<
                      endl;
567                 sequential_training_mode(cout);
568                 cout << "TESTING" << endl;
569                 testing(test, E_test_av, test_answers, cout);
570             }
571             results += test_answers.proportion_of_correct_answers();
572             cout << "Average Error Energy = " << E_test_av << endl << "Part of
                    correct = " << test_answers.proportion_of_correct_answers() <<
                   "\ttotal correct = " << test_answers.correct << "\ttotal wrong =
                    " << test_answers.wrong << endl;
573             forget_what_was_taught();
574             x = old_x;
575         }
576         return results / crossvalidate_groups.size();
577     }
```

Приложение 4. Реализация структуры нейронной сети и её процесса обучения.

Подключённые библиотеки:

```
#include "MATLAB\MATLAB.h"
#include"PCA_LDA_alglib.h"
#include <iostream>
#include<string>
#include <algorithm>
#include<vector>
#include<fstream>
#include <math.h>
#include <time.h>
```

Для реализации LDA и PCA на C++ была использована библиотека Alglib. Файл `PCA_LDA_alglib.h` играет роль моста между реализованной нейронной сетью и данной библиотекой. В файле `MATLAB.h` были реализованы функции, упрощающие создание *.m файлов в соответствии с синтаксисом MATLAB (для вывода матриц, векторов или построения графиков).

Сигмоидальная функция активации:

```
double sigmoid_logist(const double & v_induced_field,const double & sigmoid_lean)       //сигмоидальная функция [0,1]
{
    return (1.0 / (1.0 + exp(-v_induced_field*sigmoid_lean)));
}
```

Один из основных классов – персептрон:

```
class Perceptron
{
public:
    vector<double> w;           //веса
    vector<double> w_opt;       //оптимальные веса
    vector<double> delta_perv;  //предыдущее изменение веса (необходимо для эффекта инерции)
    double bias;                //порог
    double v_induced_field;     //индуцированное поле
    double y;                   //выход
```

Методы класса:

```
    //реакция перцептрона на сигнал
    double output(const vector<double> & inp,const double & sigmoid_lean)

    {
        v_induced_field = 0;
        for (int i = 0; i < w.size(); i++)                                  //     суммирование
```

```
35          {
36              v_induced_field += inp[i] * w[i];
37          }
38          v_induced_field += bias;
39          y = sigmoid_logist(v_induced_field, sigmoid_lean);              //
              активационная функция
40          return y;
41      }
42      //корректировка весов с помощью функции обратного распространения ошибки
43      vector<double> error_correction(const double & sigmoid_lean, const double
          & learning_rate,const double & inertia, const double & _error, const
          vector<double> & _y)
44      {
45          double delta = sigmoid_lean*y*(1 - y)*_error;
46          vector<double> error(w.size());                                 //
              для передачи предыдущему слою
47          bias += inertia*delta_perv.back() + learning_rate*delta;        //
              изменяем bias так же как и другие веса, но с учетом сигнала = 1
48          delta_perv.back() = inertia*delta_perv.back() + learning_rate*delta;
49          for (int j = 0; j < w.size(); j++)
50          {
51              error[j] = delta*w[j];                                      //
                  в передаче ошибки предыдущим слоям и их нейронам участвует w[n]
52              double del_w_n(inertia*delta_perv[j] + learning_rate*delta*_y[j]);
53              w[j] += del_w_n;                                            //
                  w[n+1] добавлен эффект инерции
54              delta_perv[j] = del_w_n;
55          }
56          return error;
57      }
58      //сохранение оптимальных весов
59      void record_optimal_weight()
60      {
61          w_opt = w;
62      }
63      //восстановление весов из сохраненных оптимальных
64      void recover_optimal_weight()
65      {
66          w = w_opt;
67      }
68      //забыть все, чему обучен перцептрон
69      void forget_what_was_taught()
70      {
71          bias = init_weight * ((double)rand() / RAND_MAX);
72          for (int i = 0; i < w.size(); i++)
73          {
74              w[i] = init_weight * ((double)rand() / RAND_MAX);
75          }
76      }
```

```
77      //конструктор с известными (предположительными) стартовыми весами
78      Perceptron(const vector<double> & _w)
79      {
80          w = _w;
81          delta_perv.resize(w.size() + 1);            //+1 так как в последнем
              хранится изменение bias
82          bias = init_weight * ((double)rand()/ RAND_MAX);
83      }
84      //конструктор со случайной генерацией стартовых весов
85      Perceptron(const int & w_size)
86      {
87          bias = init_weight * ((double)rand()/ RAND_MAX);
88          w.resize(w_size);
89          for (int i = 0; i < w_size; i++)
90          {
91              w[i] = init_weight * ((double)rand()/ RAND_MAX);
92          }
93          delta_perv.resize(w_size+1);                //+1 так как в последнем
              хранится изменение bias
94      }
95      ~Perceptron()
96      {
97      }
98  };
```

Ещё один из основных классов – слой:

```
100 class Layer
101 {
102 public:
103     vector<double> inp;             //текущий вектор входов
104     vector<Perceptron> perc;        //перцептроны слоя
105     vector<double> out;             //текущий вектор выходов
```

Методы класса:

```
106     //реакция слоя на сигнал
107     vector<double> output(const vector<double> &_inp, const double &
          _sigmoid_lean)
108     {
109         inp = _inp;
110         for (int k = 0; k < out.size(); k++)
111         {
112             out[k] = perc[k].output(inp, _sigmoid_lean);
113         }
114         return out;
115     }
```

```
116     //корректировка весов с помощью метода обратного распространения ошибки
117     vector<double> error_correction(const vector<double> &_error, const double
           & _sigmoid_lean, const double & _learning_rate, const double &
          _inertia, const int & size_of_the_previous_layer) //имеется ввиду слоя
          слева от текущего
118     {
119         vector<double>error(size_of_the_previous_layer);
120         for (int j = 0; j < perc.size(); j++)
121         {
122             //получаем ответ от каждого нейрона текущего слоя в виде вектора
                delta*w[i], где i - индекс нейрона предшествующего слоя
123             vector<double> perc_error(perc[j].error_correction(_sigmoid_lean,
                _learning_rate, _inertia, _error[j], inp));
124             for (int i = 0; i < error.size(); i++)
125             {
126                 error[i] += perc_error[i];          //накопление ошибки для
                    передачи в предыдущий слой
127             }
128         }
129         return error;
130     }
131     //сохранение оптимальных весов
132     void record_optimal_weight()
133     {
134         for (int i = 0; i < perc.size(); i++)
135         {
136             perc[i].record_optimal_weight();
137         }
138     }
139     //восстановление весов из сохраненных оптимальных
140     void recover_optimal_weight()
141     {
142         for (int i = 0; i < perc.size(); i++)
143         {
144             perc[i].recover_optimal_weight();
145         }
146     }
147     //забываем все значения весов
148     void forget_what_was_taught()
149     {
150         for (int i = 0; i < perc.size(); i++)
151         {
152             perc[i].forget_what_was_taught();
153         }
154     }
```

```
155     //конструктор со случайной генерацией стартовых весов
156     Layer(const int & perc_number, const int & size_of_the_previous_layer)
157     {
158
159         out.resize(perc_number);
160         for (int i = 0; i < perc_number; i++)
161         {
162             perc.push_back(Perceptron(size_of_the_previous_layer));
163         }
164     }
165     //конструктор с известными (предположительными) стартовыми весами
166     Layer(const int & perc_number, const vector<vector<double> > &
          start_weights)
167     {
168         out.resize(perc_number);
169         for (int i = 0; i < perc_number; i++)
170         {
171             perc.push_back(Perceptron(start_weights[i]));
172         }
173     }
174     ~Layer() {}
175 };
```

Вспомогательные классы «ответ» и «ответы» для обработки откликов сети и сбора статистики касательно верности полученных ответов:

```
176 class Answer
177 {
178 public:
179     int sample_id;
180     int from_net;
181     double confidence;
182     int desired;
183     Answer(const vector<double> & _from_net, const vector<double> & _expected)
184     {
185         int max_id_net = 0;
186         double max_net = INT_MIN;
187         int max_id_exp = 0;
188         double max_exp = INT_MIN;
189         double summ = 0;
190         for (int i = 0; i < _from_net.size(); i++)
191         {
192             summ += _from_net[i];
193             if (_from_net[i]>max_net)
194             {
195                 max_net = _from_net[i];
196                 max_id_net = i;
197             }
198             if (_expected[i] > max_exp)
199             {
200                 max_exp = _expected[i];
201                 max_id_exp = i;
202             }
```

```
203         }
204         confidence = _from_net[max_id_net] / summ;
205         from_net = max_id_net;
206         desired = max_id_exp;
207     }
208     bool correct()
209     {
210         if (from_net == desired)
211             return true;
212         else
213             return false;
214     }
215     ~Answer(){}
216 private:
217 };
218 class Answers
219 {
220 public:
221     vector<Answer> answers;
222     int correct;
223     int wrong;
224     double average_confidence;
225     double average_confidence_correct;
226     double average_confidence_wrong;
227     Answers()
228     {
229         correct = 0;
230         wrong = 0;
231     }
232     void add_answer(Answer & new_answer)
233     {
234         average_confidence *= answers.size();
235         average_confidence += new_answer.confidence;
236         if (new_answer.correct())
237         {
238             average_confidence_correct *= correct;
239             average_confidence_correct += new_answer.confidence;
240             correct++;
241             average_confidence_correct /= correct;
242         }
243         else
244         {
245             average_confidence_wrong *= wrong;
246             average_confidence_wrong += new_answer.confidence;
247             wrong++;
248             average_confidence_wrong /= wrong;
249         }
250         answers.push_back(new_answer);
251         average_confidence /= answers.size();
252     }
```

```
253     void clear()
254     {
255         average_confidence = 0;
256         average_confidence_correct = 0;
257         average_confidence_wrong = 0;
258         correct = 0;
259         wrong = 0;
260         answers.clear();
261     }
262     double proportion_of_correct_answers()
263     {
264         return (double)correct/(correct+wrong);
265     }
266     ~Answers(){}
267
268 private:
269
270 };
```

И, наконец, самый основной класс – нейронная сеть:

```
272 class MyNet
273 {
274 private:
275     vector<int> net_architecture;              //размерности слоев
276     vector<Layer> layers;                      //слои нейронный сети
277     double sigmoid_lean;                       //коэффициент наклона
          сигмоидальной функциии
278     double learning_rate;                      //скорость обучения
279     double inertia;                            //инерция обучения
280     int epoch;                                 //текущая эпоха
281     int max_epoch;                             //максимальное количество эпох
282     bool fixed_time;                           //ограниченно ли время для
          тренировки
283     double max_time;                           //если время ограничено, то
          это - значение ограничения времени в минутах
284     double E_av_min;                           //минимальная достигнутая
          средняя энергия ошибки на валидационном множестве (используется для
          сохранения оптимальных весов)
285     vector<double> E;                          //энергия ошибки на каждой
          итерации
286     vector<double> E_train_av;                 //средняя энергия ошибки по
          всему обучающему множеству на каждой эпохе
287     vector<double> E_valid_av;                 //средняя энергия ошибки на
          валидационном множестве
288     void error_correction(vector<double> network_output, vector<double>
          expected_output)        //обратное распространение ошибки
289     {
290         vector<double> error(expected_output);
291         for (int k = 0; k < error.size(); k++)
292         {
293             error[k] -= network_output[k];                          //error =
                expected_output - network_output
294             E.back() += error[k] * error[k];
295         }
```

```
296         E.back() /= 2;                                          // текущая энергия ошибки
297         E_train_av.back() += E.back();                          // накапливаем ее в среднюю, после чего разделим на n
298         for (int i = layers.size() - 1; i >-1; i--)             //обратное распространение ошибки от слоя к слою
299         {
300             error = layers[i].error_correction(error, sigmoid_lean, learning_rate, inertia, net_architecture[i]);
301         }
302     }
303     //в случае если на данный момент ошибка наименьшая - сохраняем оптимальные веса
304     void record_optimal_weight()
305     {
306         for (int l = 0; l < layers.size(); l++)
307         {
308             layers[l].record_optimal_weight();
309         }
310     }
311     //откат к наиболее оптимальным весам
312     void recover_optimal_weight()
313     {
314         for (int l = 0; l < layers.size(); l++)
315         {
316             layers[l].recover_optimal_weight();
317         }
318     }
319     //сохранение весов в случае если текущая энергии ошибки минимальна на данный момент
320     double fix_weights_if_E_min(const double &E_av)
321     {
322         if (E_av_min > E_av)            //отслеживаем минимальное значение средней энергии ошибки (на валидационном множестве)
323         {
324             double delta = E_av_min - E_av;
325             E_av_min = E_av;
326             cout << "Eav validate min = " << E_av_min << "\tepoch = " << epoch << endl;
327             E_av_min_epoch = epoch;
328             record_optimal_weight();    //в случае, если на данный момент ошибка наименьшая - сохраняем оптимальные веса
329             return delta;
330         }
331         else
332         {
333             return 0;
334         }
335     }
```

```
336     //анализ условий для прекращения тренировки
337     int exit_analysis_of_average_error(ostream &out)
338     {
339         E_valid_av.push_back(0);
340         testing(validate, E_valid_av.back(), validate_answers, out);
341         cout << "accuracy on the validation set: " <<
              validate_answers.proportion_of_correct_answers() << endl;
342         if (fix_weights_if_E_min(E_valid_av.back()) > delta_validate_Eav_min)
                    //если текучщее значение энергии ошибки минимально
343         {
344             number_of_failed_epochs = 0;
345             exit_conditions = false;
346             if (E_valid_av.back() < delta_validate_Eav_min)                // 
                  заканчиваем обучение, если достигли нужной погрешности
347             {
348                 exit_conditions = true;
349             }
350         }
351         else
352         {
353             number_of_failed_epochs++;
354             if (number_of_failed_epochs == critical_number_of_failed_epochs)
355             {
356                 exit_conditions = true;
357             }
358         }
359         if (epoch > 0)
360             out << "Delta = " << E_valid_av.back() - E_valid_av
                      [E_valid_av.size() - 2] << endl;
361         validate_answers.clear();
362         if (epoch < 25)                     //предохранитель от слишком
              раннего останова
363             return false;
364         if (exit_conditions)
365         {
366             recover_optimal_weight();       //откат к наиболее оптимальным
                  весам
367             return true;
368         }
369         else
370         {
371             return false;
372         }
373     }
375 public:
376     vector<vector<double> > x;                  //все имеющиеся входные образы
377     vector<vector<double> > targets;            //все выходные сигналы для
          образов
378     vector<vector<int> > crossvalidate_groups;  //индексы разбиения на группы
          для кроссвалидации
379     vector<int> train;                          //индексы тренировочных
          образов
380     vector<int> validate;                       //индексы валидационных
          образов
381     vector<int> test;                           //индексы тестовых образов
```

```
382     int W;                                      //общее количество весов
383     double delta_validate_Eav_min;              //граница требуемой ошибки
384     int E_av_min_epoch;                         //
385     double E_test_av;                           //средняя энергия ошибки на
          тестовом множестве
386     Answers test_answers;                       // <ответ полученный от сети,
          желаемый ответ>
387     Answers validate_answers;                   //
388     Answers train_answers;                      //
389     int number_of_failed_epochs;                //количество эпох подряд, при
          которых не уменьшено минимальное значение энергии ошибки
390     int critical_number_of_failed_epochs;       //максимальное количество
          "неудачных" эпох подряд, то есть тех, которые не уменьшили энергию
          ошибки
391     bool exit_conditions;                       //переменная сигнала остановки
           обучения

393     //подача на вход тренировочного примера и корректировка весов с помощью
          обратного распространения ошибки
394     void net_train(const vector<double> & input, const vector<double> &
          desired_output, Answers &answers)
395     {
396         vector<double>temp_input_output(input);
397         for (int l = 0; l < layers.size(); l++)
398         {
399             temp_input_output = layers[l].output(temp_input_output,
                sigmoid_lean);  //передача входного сигнала от слоя к слою
400         }
401         answers.add_answer(Answer(temp_input_output, desired_output)); //
              записываем полученные ответы
402         E.push_back(0);                                                 //
              добавляем в конец вектора, в котором находятся значения текущей
              энергии ошибки, новый нулевой элемент
403         error_correction(temp_input_output, desired_output);
404     }

405     //подача на вход тестового одного тестового примера и запись ответов сети
406     void net_test(const vector<double> & input,const vector<double> &
          desired_output, double &E_av, Answers &answers)
407     {
408         vector<double>temp_input_output(input);                         //
              берем  входной сигнал
409         for (int l = 0; l < layers.size(); l++)
410         {
411             temp_input_output = layers[l].output(temp_input_output,
                sigmoid_lean);  //передача входного сигнала от слоя к слою
412         }
413         answers.add_answer(Answer(temp_input_output, desired_output));  //
              записываем ответы
414         vector<double> error(desired_output);
415         for (int k = 0; k < error.size(); k++)
416         {
417             error[k] -= temp_input_output[k];                           //
                  error = expected_output - network_output
418             E_av += error[k] * error[k];
419         }
420     }
```

```
421     //процедура последовательной подачи на вход всего тестового множества
422     void testing(const vector<int> &test_id, double &E_av, Answers &answers,
          ostream &out)
423     {
424         unsigned int start_time = clock();
425         for (int v = 0; v < test_id.size(); v++)
426         {
427             net_test(x[test_id[v]], targets[test_id[v]], E_av, answers);
428         }
429         unsigned int end_time = clock();
430         out << endl << "average time for one test: " << double(end_time -
              start_time) / test_id.size() / CLOCKS_PER_SEC << " sec" << endl;
431         E_av /= 2 * test_id.size();
432         out<<"Average Error Energy = "<<E_av<<endl<<"Part of correct =
              "<<answers.proportion_of_correct_answers()<<"\ttotal correct =
              "<<answers.correct<<"\ttotal wrong = "<<answers.wrong<<endl;
433         out<<"answer:       ";
434         for (int i = 0; i < answers.answers.size(); i++)
435         {
436             out.width(5);
437             if (!answers.answers[i].correct())
438                 out<<answers.answers[i].from_net;
439         }
440         out<<endl<<"right answer:";
441         for (int i = 0; i < answers.answers.size(); i++)
442         {
443             out.width(5);
444             if (!answers.answers[i].correct())
445                 out<<answers.answers[i].desired;
446         }
447         out << endl << "confidence:  ";
448         for (int i = 0; i < answers.answers.size(); i++)
449         {
450             out.width(5);
451             out.precision(2);
452             if (!answers.answers[i].correct())
453                 out << answers.answers[i].confidence;
454         }
455         out<<endl;
456         out << "average confidence:  " << answers.average_confidence << endl
              << "confidence in correct:  " << answers.average_confidence_correct
              << endl << "confidence in wrong:  " <<
              answers.average_confidence_wrong << endl;
457     }
458     //последовательный режим обучения
459     void sequential_training_mode(ostream &out)
460     {
461         out << "start training:" << endl;
462         unsigned int start_time = clock();
463         E_av_min = INT_MAX;
464         exit_conditions = false;
465         for (epoch = 0; epoch < max_epoch; epoch++)         //выход при
              условии исчерпания количества эпох,
```

```
466         {                                                   //либо если
              средняя энергия ошибки опустится ниже нужного уровня
467           random_shuffle(train.begin(), train.end());     //перемешиваем
                порядок подачи тренировочных элементов для предотвращения
                замкнутых циклов в эволюции
468           out << "epoch " << epoch + 1 << endl;
469           unsigned int start_time_2 = clock();
470           E_train_av.push_back(0);                        //добавляем в
                конец вектора, в котором находятся значения средней энергии
                ошибки за эту эпоху, новый нулевой элемент
471           for (int n = 0; n < train.size(); n++)          //итерация (такт
                времени), соответствует n-му обучающему образу
472           {
473               net_train(x[train[n]], targets[train[n]], train_answers);   //
                    берем n-ый тренировочный входной сигнал и подаем его на вход
                     сети в обучающем режиме
474           }
475           //до этого в течение эпохи мы суммировали текущие значения ошибки,
                 теперь мы делим на количество элементов, чтобы вычислить
                среднее значение
476           E_train_av.back() /= train.size();
477           unsigned int end_time_2 = clock();
478           if (epoch == 0)
479           {
480               if (fixed_time)
481               {
482                   max_epoch = max_time * 60 / (double(end_time_2 -
                      start_time_2) / CLOCKS_PER_SEC);
483                   out << "new max epoch = " << max_epoch << endl;
484                   cout << "new max epoch = " << max_epoch << endl;
485               }
486           }
487           out << "train Average Error Energy = " << E_train_av.back() <<
                endl << "Part of correct = " <<
                train_answers.proportion_of_correct_answers()
488             << "\ttotal correct = " << train_answers.correct << "\ttotal
                  wrong = " << train_answers.wrong << endl;
489           cout << "train Average Error Energy = " << E_train_av.back() <<
                endl << "Part of correct = " <<
                train_answers.proportion_of_correct_answers()
490             << "\ttotal correct = " << train_answers.correct << "\ttotal
                  wrong = " << train_answers.wrong << endl;
491           epoch > 0 ? out << "average Error energy = " << E_train_av.back()
                << " \tDelta = " << E_train_av.back() - E_train_av
                [E_train_av.size() - 2] << endl
492             << "time of training: " << double(end_time_2 - start_time_2) /
                   CLOCKS_PER_SEC << " sec" << endl :
493             out << "average Error energy = " << E_train_av.back() << endl
                  << "time of training: " << double(end_time_2 -
                  start_time_2) / CLOCKS_PER_SEC << " sec" << endl;
494           train_answers.clear();
```

```
495             if (exit_analysis_of_average_error(out))                    //остановка обучения в случае переобучения или достижения нужной погрешности
496             {
497                 break;
498             }
499         }
500         recover_optimal_weight();
501         unsigned int end_time = clock(); //конечное время
502         out << "min average E = " << E_av_min << endl;
503         out << "total time of training: " << double(end_time - start_time) / CLOCKS_PER_SEC << " sec" << endl;
504         cout << "min average E = " << E_av_min << endl;
505         cout << "total time of training: " << double(end_time - start_time) / CLOCKS_PER_SEC << " sec" << endl;
506     }

517     void forget_what_was_taught()
518     {
519         test.clear();
520         validate.clear();
521         train.clear();
522         E.clear();                          //энергия ошибки на каждой итерации
523         E_train_av.clear();                 //средняя энергия ошибки по всему обучающему множеству на каждой эпохе
524         E_valid_av.clear();                 //средняя энергия ошибки на валидационном множестве
525         E_test_av = 0;                      //средняя энергия ошибки на тестовом множестве
526         test_answers.clear();               // <ответ полученный от сети, желаемый ответ>
527         validate_answers.clear();           //
528         train_answers.clear();              //
529         for (int i = 0; i < layers.size(); i++)
530         {
531             layers[i].forget_what_was_taught();
532         }
533     }

587     void new_net_architecture(const vector<int> & _net_architecture)
588     {
589         net_architecture = _net_architecture;
590         layers.clear();
591         W = 0;                                          //подсчет общего количества весов
592         layers.push_back(Layer(net_architecture[1], net_architecture[0]));
593         for (int i = 1; i < net_architecture.size() - 1; i++)
594         {
595             layers.push_back(Layer(net_architecture[i + 1], net_architecture[i]));
596             W += (net_architecture[i - 1] + 1) * net_architecture[i];
597         }
598         cout << "network configuration completed" << endl;
599 
600     }
```