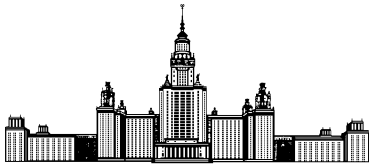
Московский государственный университет имени М. В. Ломоносова



Факультет Вычислительной Математики и Кибернетики
Кафедра Системного Программирования

КУРСОВАЯ РАБОТА СТУДЕНТА 341 ГРУППЫ

Реализация алгоритмов частичного обучения на Apache Spark

Выполнил:
студент 3 курса 341 группы
Кемаев Юрий Юрьевич

Научный руководитель:
Астраханцев Никита Александрович

Москва, 2016

Содержание

Аннотация

Последние годы особую популярность набирает фреймворк для распределенной обработки данных *ApacheSpark*, используемый в задачах анализа данных. В частности, быстро развивается библиотека машинного обучения *MLLib*. В данной работе будут проанализированы распространенные алгоритмы частичного обучения и реализованы некоторые из них, отсутствующие в *MLLib*, а именно: *Semisupervised Expectation Maximization*, *Self-Training*, *Co-Training*.

Введение

Данная работа выполнена в контексте задач машинного обучения и быстрой обработки данных. В связи с чем автор намеренно опускает большую часть технических деталей устройства используемых в работе инструментов, оставляя лишь информацию, необходимую для понимания предметной области и осознания концепций, на которых основана данная курсовая работа.

Ключевые понятия данной работы — *частичное обучение*. Главное его свойство, обеспечивающее преимущество в современных условиях анализа данных, — способность тренировать достаточно эффективные модели алгоритмов классификации и регрессии на малых выборках *размеченных* данных при условии, что в момент обучения доступны все *неразмеченные* данные.

Задачи машинного обучения[1]

Машинное обучение (*Machine Learning*) — обширный подраздел искусственного интеллекта, математическая дисциплина, использующая разделы математической статистики, численных методов оптимизации, теории вероятностей, дискретного анализа, и извлекающая знания из данных[2].

В последние годы данная область стремительно развивается, и в настоящее время задачи машинного обучения возникают в таких областях, как: банковское дело, медицина, распознавание образов, текста, речи и др.

Традиционно задачи машинного обучения делятся на **2 типа**: **обучение без учителя** (*unsupervised*) и **обучение с учителем** (*supervised*). Также выделяют еще один тип задач, называемый задачами **частичного обучения** (*semi-supervised learning*), которому и посвящена данная работа.

Обучение без учителя

Пусть $\mathbf{X} = \{x_1, \ldots, x_n\}$ - множество из n точек, где $x_i \in \mathcal{X}\ \forall i \in \overline{1..n}$. Предполагается, что все точки независимы и распределены по одному закону на множестве $\mathcal{X}$.

Цель задачи обучения без учителя — обнаружить скрытую структуру данных $\mathbf{X}$.

Обучение с учителем

Дано множество $\mathbf{X}$, элементами которого являются пары (x_i, y_i), где $x_i \in \mathcal{X}, y_i \in \mathcal{Y}\ \forall i \in \overline{1..n}$. Предполагается, что все точки независимы и распределены по одному закону на множестве $\mathcal{X} \times \mathcal{Y}$.

Элементы $y_i \in \mathcal{Y}$ называются **метками** (*labels*) точек $x_i \in \mathcal{X}$. Множество размеченных точек $\mathbf{X}$ называется **тренировочным**, а неразмеченных - **тестирующим**.

---

[1] Подробное описание в работе *O. Chapelle* и *B. Schlkopf "Semi-Supervised Learning"* [5]

[2] en.wikipedia.org/wiki/Machine_learning

Цель задачи обучения с учителем — найти наилучшее по некоторому критерию отображение $\mathcal{X} \to \mathcal{Y}$.

Если $\mathcal{Y}$ - конечное множество, то дана **задача классификации**.

Напротив, если $\mathcal{Y}$ - континуальное множество, то дана **задача регрессии**.

Трансдуктивное и индуктивное обучение

Существуют **2** принципиально разных **метода** машинного обучения: **индуктивное** и **трансдуктивное**.

Индуктивное обучение характеризуется тем, что на *тренировочной* выборке обучается модель машинного обучения, способная работать с любыми данными из исходного пространства. Таким образом, данный подход можно описать как "переход от частного к общему".

Понятие **трансдуктивное обучение** введено *V. Vapnik* в работе *"Statistical Learning Theory"* [8].

В отличие от индуктивного обучения, этот подход можно охарактеризовать как "рассуждения о частных случаях (тестовых данных) на основании частных случаев (тренировочных данных)". В рамках данного подхода на выходе имеется модель машинного обучения, способная предсказывать лишь те данные, которые были использованы в ходе её обучения (*тренировочные и тестовые выборки*).

Частичное обучение

Данный тип обучения включает в себя черты как обучения с учителем, так и без учителя.

Задачи данного типа обычно характеризуются тем, что на входе имеются размеченная тестирующая $\mathbf{L}$ и тренировочная $\mathbf{U}$ выборки, причём $|\mathbf{L}| \ll |\mathbf{U}|$.

Важно отметить, что данный тип задач чаще всего встречается на практике, так как обычно:

1. Для хороших моделей требуется большой объем размеченных данных
2. Разметка данных требует вложений ресурсов (людских, денежных, временны́х)
3. Размеченные данных мало, а неразмеченных — очень много

Частичное обучение отчасти решает эти проблемы, так как для него требуется сравнительно небольшое количество размеченных данных (в исследовании *Zhu X.* [9] утверждается, что в некоторых случаях достаточно **1**-й размеченной точки от каждого класса) для обучения модели и по качеству моделей сравнимо с традиционными типами обучения. Разумеется, времени на такое обучение требуется больше.

В основе алгоритмов частичного обучения лежат следующие предложения:

1. *Размеченные данные состоятельны по доступным признакам*
2. *Размеченные данные несмещены относительно неразмеченных*

3. *Имеется хотя бы один представитель каждого класса в размеченных данных*

Если хотя бы одно из перечисленных предположений не выполняется, результат работы алгоритмов будет непредсказуемым (см. исследование *Zhu X.* [5]).

В последующих разделах работы будут рассмотрены некоторые классы алгоритмов частичного обучения и их характерные представители.

Фреймворк Apache Spark

В последние годы в связи с многократным увеличением объема требующих обработки данных[1] получил широкое распространение подход, основанный на распределенной обработке данных.

В частности, в настоящее время широко используются фреймворки, работающие в рамках модели *MapReduce*. Эта модель позволяет пользователям писать распределенные приложения не задумываясь о технических деталях, таких как устранение сбоев или распределение нагрузки.

Основной недостаток данной модели, описанный в статье *Zaharia M. и др.* [4], – отсутствие поддержки распределенной памяти. В применении к задачам анализа данных это становится критическим фактором, по причине которого *MapReduce* не используется в указанной области.

Среди задач анализа данных можно выделить **2** основных семейства:

- Итеративные задачи

  Бóльшая часть алгоритмов машинного обучения применяют операции к одними и теми же данными много раз с целью оптимизиции нужных параметров и характеристик. *MapReduce* на каждой итерации требует новой загрузки данных с диска, что критически увеличивает время работы алгоритма.

- Задачи, требующие интерактивный анализ данных

  В ходе такого анализа происходят многочисленные запросы чтения к хранимым данным. Модель *MapReduce* в этом случае требует для каждого запроса полную загрузку данных с дисков, что также неприемлемо с точки зрения быстродействия.

Фреймворк *Apache Spark* решает проблемы повторной загрузки даных с диска и позволяет решать описанные выше задачи эффективным образом. Как именно — описано в соответствующем разделе данной работы.

[1] *The rapid growth of Global Data* assets1.csc.com/insights/downloads/CSC_Infographic_Big_Data.pdf

1 Постановка задачи

Цель данной работы — **реализация некоторых алгоритмов частичного обучения на *Apache Spark***.

Этапы работы:

1. Анализ существующих алгоритмов частичного обучения с целью выявления наиболее эффективных и применимых на практике в задачах классификации
2. Выбор и реализация нескольких алгоритмов для задач бинарной классификации на *Apache Spark* с использованием *MLlib*
3. Выбор метрик качества и сравнительный анализ работы реализованных алгоритмов

2 Обзор существующих решений

В данной главе рассмотрены наиболее распространенные алгоритмы частичного обучения, подробно описаны их достоинства и недостатки.

Также освещены основные концепции и принципы работы фреймворка для распределенной обработки данных *Apache Spark*. В частности, подробно рассмотрена его библиотека для машинного обучения *MLLib*.

Дополнительно описаны метрики для измерения качества работы алгоритмов частичного обучения, реализованные в *MLLib*.

2.1 Алгоритмы частичного обучения

2.1.1 Expectation Maximization (EM)

EM, согласно работе *O. Chapelle* и *B. Schlkopf "Semi-Supervised Learning"* [5], является первым алгоритмом частичного обучения.

EM строит модель исходя из следующих предположений:

1. Совместная вероятность для модели $p(x, y) = p(y) \cdot p(x|y)$, где $p(x|y)$ - распределение смеси (например, гауссиан)

2. Смесь должна быть **распознаваемой** (*identifiable*), что подразумевает, что за конечное число наблюдений этой смеси возможно полностью восстановить её параметры

При истинности этих предположений гарантируется, что качество будет выше традиционного *supervised EM*.

Основная идея — *обнаружить значения скрытых параметров базовых распределений смеси в соответствии с принципом максимального правдоподобия.*

EM алгоритм относится к классу генеративных моделей

Алгоритм 1: EM

Вход: L, U, $Classifier$;
Выход: $Model$;
  $Model := TrainModel(L)$; {Построить модель по размеченным данным}
  повторять
    $P :=$ оценки $Model$ метки классов для каждого $x_i \in U$; {E-шаг}
    $Model :=$ TrainModel$(L \cup (U,\ P))$; {M-шаг}
  пока не сойдется

2.1.2 Self-Training

Self-training — один из самых ранних и часто используемых в задачах частичного обучения алгоритм.

Основная идея — *обучаться, предсказывать и затем переобучаться, используя свои собственные наиболее надежные предсказания.*

Self-training относится к wrapper-алгоритмам

Алгоритм 2: Self-training

Вход: L, U, $Classifier$;
Выход: L', U';
  $L' := L$; {инициализация}
  $U' := U$;
  повторять
    обучить $Classifier$ на данных L'
    $P :=$ предсказания $Classifier$ на U';
    $N_L :=$ извлечь из P пары, у которых $confidence > threshold$
    $L' := L' \cup N_L$;
    $U' := U' \setminus N_L$;
  пока $|U'| > 0$ и $|N_L| > 0$

2.1.3 Co-training

Co-training был впервые описан в работе *Blum A.* и *Mitchell T. "Combining labeled and unlabeled data with co-training"* [2].

Этот алгоритм дополнительно основан на предположениях, что:

1. Признаки точек можно разделить на 2 условно независимых по классам множества

2. Каждое из этих множеств достаточно хорошо представляет исходные точки

Основная идея — разбить данные на 2 независимых репрезентативных множества, каждому из них поставить в соответсвие определенный алгоритм, далее алгоритмы поочередно обучают друг друга.

Основная цель — *добиться статистической независимости между ответами алгоритмов и размечать точки с наибольшей уверенностью предсказания.*

Co-training относится к wrapper-алгоритмам

2.1.4 Multi-view Learning

Multi-view Learning естественным образом обобщает *Co-training*.

Основная идея — та же, что и у *Co-training*, но теперь используются $N > 2$ алгоритмов. Большое количество независимых алгоритмов должны соглашаться на неразмеченной точке, чтобы она была добавлена в обучающую выборку.

Алгоритм 3: Co-training

Вход: L, U, $Classifier_1$, $Classifier_2$;
Выход: L', U';
разделить $L \cup U$ по признакам точек на 2 множества $L_1 \cup U_1$ и $L_2 \cup U_2$;
повторять
обучить $Classifier_i$ на данных L_i для i=1,2;
$P_i :=$ предсказания $Classifier_i$ на U_i для i=1,2;
$NL_i :=$ извлечь из P_i пары, у которых $confidence > threshold$ для i=1,2;
$L_1 := L_1 \cup NL_2$;
$L_2 := L_2 \cup NL_1$;
$U_1 := U_1 \setminus NL_2$;
$U_2 := U_2 \setminus NL_1$;
пока $|U_1| + |U_2| > 0$ и $|NL_1| + |NL_2| > 0$
$L' := L_1 \cup L_2$;
$U' := U_2 \cap U_1$;

2.1.5 Cluster and Label Approach

Cluster and Label Approach — естественная комбинация обучения с учителем и без.

Основная идея — кластеризовать все данные, затем обучить модель на каждом кластере и разметить с помощью неё оставшиеся в кластере точки.

Необходимые условия:

1. Кластера согласуются с истинными разделяющими поверхностями
2. В каждом кластере есть хотя бы одна размеченная точка

При невыполнении этих условий качество модели будет низким.

Cluster and Label Approach относится к wrapper-алгоритмам

Алгоритм 4: CLA

Вход: L, U, $Clusterizator$, $Classifier$;
Выход: L';
$L' := \varnothing$;
кластеризовать $L \cup U$ по K кластерам, используя $Clusterizator$;
для каждого кластера $k \in K$
$L' := L' \cup$ метки из k;
обучить $Classifier$ на размеченных данных из k;
$NL :=$ метки, полученные с помощью $Classifier$, для оставшихся данных;
$L' := L' \cup NL$;
end для

2.1.6 Transductive support vector machine (T-SVM)

T-SVM — модифицированный для работы с неразмеченными данными традиционный *SVM*.

Основная идея — построить линейную разделяющую границу с максимальным отступом (*margin*) в преобразованном по некоторому *ядру* исходном пространстве, используя как размеченные, так и неразмеченные точки.

Это можно интерпретировать в том смысле, что к функция штрафа оригинального *SVM* добавляется новое слагаемое, штрафующее за малый отступ от неразмеченных данных:

$$J(w) = \underbrace{\sum_{i=1}^{N}(1-m_i)_+ + \frac{1}{2C}||w||^2}_{original} + \underbrace{\lambda \sum_{i=N+1}^{N+U}(1-|m_i|)_+}_{semi-supervised}$$

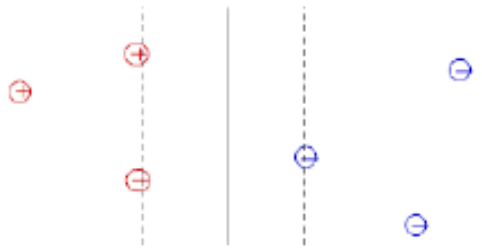


Рис. 1: Традиционный SVM (с презентации В. Гулина, Техносфера)

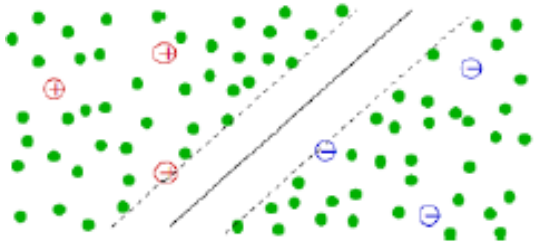


Рис. 2: Semi-supervised SVM (с презентации В. Гулина, Техносфера)

TSVM относится к алгоритмам, избегающим регионов с высокой плотностью (*Avoiding Changes in Dense Regions*)

Псевдокод оригинального *SVM* можно найти в работе *V. Vapnik "Statistical Learning Theory"* [8]

2.1.7 Semi-Supervised Trees

Алгоритм **Semi-Supervised Trees** является модификацией *Decision Trees*. Отличие заключается в том, что она использует информацию о неразмеченных данных. Описанный ниже алгоритм был впервые рассмотрен в работе *A. Criminisi и др. "Decision forests for classification, regression, density estimation, manifold learning and semi-supervised learning"* [1].

Основная идея — приближать распределение исходных данных смесью гауссиан с помощью леса решающих деревьев, гарантирующих локально минимальную энтропию. Использование деревьев означает, что итоговая модель, в отличие от *TSVM*, будет **нелинейной**.

Энтропия d-мерного нормального распределения

$$\mathbf{H}(\mathbf{X}) = \frac{1}{2}\ln(2\pi e)^d|\Sigma(X)|$$

В данной модели решающих деревьев используется следующий *information gain*:

$$\mathbf{I} = I^S + \alpha I^U,$$

$$I^S = H(X_L) - \frac{|X_L^L|}{|X_L|}H(X_L^L) - \frac{|X_L^R|}{|X_L|}H(X_L^R)$$

$$I^U = \ln|\Sigma(X)| - \frac{|X_{L\cup U}^L|}{|X_{L\cup U}|}\ln|\Sigma(X_{L\cup U}^L)| - \frac{|X_{L\cup U}^R|}{|X_{L\cup U}|}\ln|\Sigma(X_{L\cup U}^R)|$$

Соответственно, вероятность принадлежности точки x классу y определяется как

$$p(y|x) = \frac{1}{T}\sum_{t=1}^{T} p_t(y|x)$$

Алгоритм Semi-Supervised Trees относится к моделям, основанным на плотностях распределения данных

Псевдокод оригинала построения дерева можно найти на *Wikipedia*[1], а леса из деревьев в статье *Breiman L. "Random forests"* [3]

2.1.8 Gradient Boosting with Priors and Manifold regularization

Данный алгоритм описан в статье *Saffari A. "Multi-Class Semi-Supervised and Online Boosting"* [7] и является расширением градиентного бустинга.

Дополнительно к функционалу штрафа добавляются 2 регуляризатора:

1. **Prior** — из предположения, что распределение размеченных данных по выделенным кластерам должно совпадать с распределение всех данных по тем же кластерам

2. **Manifold** — из предположения, что данные лежат на некотором многообразии меньшей размерности, включенном в Евклидово пространство

Таким образом, функционал ошибки принимает вид

$$L(h) = \sum_{i=1}^{N_\mathbf{L}} L(y_i, h(x_i)) + \lambda J_{prior}(p,q) + (1-\lambda)\sum_{i=1}^{N_\mathbf{L}}\sum_{j=1}^{N_\mathbf{U}} s(x_i,x_j)J_{manifold}(x_i,x_j,h),$$

где $p(y|x)$ и $q(y|x)$ — соответственно *априорное* и *апостериорное* распределение классов по кластерам.

---

[1] en.wikipedia.org/wiki/C4.5_algorithm

[2] en.wikipedia.org/wiki/Gradient_boosting

Псевдокод можно найти на *Wikipedia*[2]

2.1.9 SemiBoost

SemiBoost — модификация **AdaBoost**, использующая информацию о неразмеченных данных. Подробно описан в статье *P. Mallapragada и др. "Semiboost: Boosting for semi-supervised learning"*[6]

Основная идея — провести серию последовательных итераций, на каждой из которых добавить в размеченные данные точки с наибольшей уверенностью предсказания и обучить на новых данных новую модель, после чего добавить её к ансамблю.

Предполагается, что:

1. Неразмеченные точки, похожие по некоторой мере $S(x_i, x_j)$ друг на друга, должны иметь одну метку класса
2. Точки, похожие по S на размеченные, должны иметь такую же метку класса

Таким образом, функционал штрафа принимает вид:

$$L(h(x)) = L_{\mathbf{L}}(y, H(x)) + \lambda L_{\mathbf{U}}(H(x)),$$

$$L_{\mathbf{L}}(y, S) = \sum_{i=1}^{N_{\mathbf{L}}} \sum_{j=1}^{N_{\mathbf{U}}} S_{ij} e^{-2y_i^l y_j^u}, \; L_{\mathbf{U}}(y_u, S) = \sum_{i=1}^{N_{\mathbf{U}}} \sum_{j=1}^{N_{\mathbf{U}}} S_{ij} cosh(y_i^u - y_j^u)$$

SemiBoost относится к алгоритмам типа "ансамбль"

Алгоритм 5: SemiBoost

Вход: L, U, $WeakLearner$, T;
Выход: ансамбль H;
вычислить матрицу похожести S между каждой парой из $L \cup U$;
$H := 0$;
для $t = 1, \ldots, T$
  $NL := \varnothing$
  для $x_i \in U$ {вычислить априорные вероятности с учетом дополнительного штрафа}
    $p_i = \sum_{j=1}^{N_{\mathbf{L}}} S_{ij} e^{-2H_i} \mathbf{I}(y_j = 1) + \frac{\lambda}{2} \sum_{j=1}^{N_{\mathbf{U}}} S_{ij} e^{H_j - H_i}$;
    $q_i = \sum_{j=1}^{N_{\mathbf{L}}} S_{ij} e^{2H_i} \mathbf{I}(y_j = -1) + \frac{\lambda}{2} \sum_{j=1}^{N_{\mathbf{U}}} S_{ij} e^{H_i - H_j}$;
    если $|p_i - q_i| > threshold$ **то**
      $NL := NL \cup (x_i, y_i)$
    end если
  end для
  обучить $WeakLearner$ на NL;
  $\epsilon_t = \frac{\sum_i^{N_{\mathbf{U}}} p_i \mathbf{I}(h_i = -1) + \sum_i^{N_{\mathbf{U}}} q_i \mathbf{I}(h_i = 1)}{\sum_i (p_i + q_i)}$;
  $\alpha_t = \frac{1}{4} \ln \frac{1 - \epsilon_t}{\epsilon_t}$;
  $H(x) := H(x) + \alpha_t h_t(x)$;
end для

2.2 Анализ существующих алгоритмов

Expectation Maximization (EM)

+ Можно использовать с любой вероятностной моделью
− Сильно зависит от начальных размеченных данных (локальность)
− Если исходные предположения не выполняются, работает плохо

Чтобы избежать неудачную инициалицию, применяют *активное обучение* для выбора информативных начальных данных.

Self-Training

+ Можно использовать с любым алгоритмом обучения с учителем
+ Простота алгоритма и неплохие результаты на практике (*Zhu X.* [9], стр. 11)
− Ошибки в размеченных данных будут многократно усилены
− Неустойчив к выбросам в данных
− Сложно анализировать в общем смысле

Чтобы избежать усиления ошибок, применяется следующий подход: на каждой итерации помимо добавления новых точек с высокой уверенностью предсказания нужно исключить из тренировочных данных точки с низкой уверенностью предсказания.

Co-Training

+ Можно использовать с любыми алгоритмами обучения с учителем (причём чем сильнее они различаются, тем лучше в итоге)
+ При выполнении предположений отличные результаты на практике (*Zhu X.* [9], стр. 12-13)
− Слишком сильные предположения для исходных данных
− Нетривиальный поиск требуемого разбиения множества признаков

Multi-view Learning

+ При выполнении предположений работает лучше *Co-Training*
− Ещё более нетривиальный поиск требуемого разбиения множества признаков , чем у *Co-Training*

Cluster and Label Approach

+ Хорошо работает в случае ярко выраженных кластеров
− Данные не всегда хорошо кластеризуются

Transduction support vector machine (T-SVM)

+ Оптимальная разделяющая поверхность
− Решение неустойчиво, елси нет зазора между классами
− Требуется настройка дополнительного параметра λ
− Работает долго, без модификаций неприменим на практике

Semi-Supervised Trees

+ Нелинейные границы разделяемых областей
+ Устойчивость к переобучению
+ Хорошая интерпретация
− Большие временные затраты (из-за матрицы ковариаций при подсчете энтропии)

Gradient Boosting wth Priors and Manifold regularization

+ Качество выше, чем у *WeakLearner*
+ Устойчивость к переобучению
− Подбор параметра λ (иногда имеет смысл делать $\lambda = \lambda(step)$)

SemiBoost

+ Качество заметно выше, чем у *WeakLearner*
+ Неявно используются *Prior*- и *Cluster* регуляризаторы
− Локальность (свойство *AdaBoost*)

2.3 Фреймворк *Apache Spark*

Фреймворк *Apache Spark* предназначен для организации распределенных вычислений на нескольких рабочих станциях и отличается от других инструментов этого семейства тем, что решает проблемы повторной загрузки даных с диска. В частности, эта особенность позволяет решать итеративные задачи и проводить анализ данных наиболее эффективным на данный момент образом.

В этом фреймворке данные структуированы в **resilient distributed dataset (RDD)** —- устойчивые к отказам и доступные только для чтения неизменяемые коллекции объектов, распределенные в памяти множества машин.

Ключевая особенность *RDD* в том, что он не требует репликации данных. Для восстановления сбоев используется несколько иной механизм. А именно, для каждого *RDD* хранится *lineage* — граф операций, в результате которых было получено текущее состояние коллекции — и при потере данных происходит определение операций, требующих восстановления состояния, и их выполнение. Для обеспечения эффективности данного подхода в *RDD* допускаются только *coarse-grained* трансформации, в которых каждая операция производится над всеми элементами коллекции. Соответственно, их противоположность, *fine-grained* операции, подразумевающие работу над частью коллекции, не поддерживаются. Авторы данной концепции хранения данных *Zaharia M. и др.* в своей работе *"Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing"* [4] утверждают, что такое ограничение не сильно сужает область применения по причине того, что приложения с параллельной обработкой данных естественным образом выполняют операции с целыми множествами объектов - иными словами, проводят в основном *coarse-grained* трансформации.

Все операции с *RDD* делятся на **трансформации** и **действия**.

Трансформация (transformation) — это операция над данными из файловой системы или *RDD*, результатом действия которой является новый *RDD*. *RDD* могут быть порождены только операциями этого типа.

Примеры трансформаций: *map*, *filter*, *join*.

Действие (action) — операция над *RDD*, возвращающая некоторое вычисленное значение либо сохраняющее коллекцию на диск.

Примеры действий: *count*, *collect*, *save*.

Необходимо отметить, что трансформации вычисляются лениво (*lazily*). Тем самым, данные не будут вычислены и загружены в память до тех пор, пока они не потребуются для каких-нибудь действий.

Программист имеет возможность указать, какие *RDD* будут использованы в будущем с помощью команды *persist*, и *Apache Spark* по возможности будет держать их в оперативной памяти.

Авторы концепции *RDD* помимо описанных выше также указывают следующие ее отличительные особенности:

- В случае сбоя восстановливается лишь необходимая часть RDD, при том это делается в параллельном режиме на нескольких узлах. Тем самым откаты всей системы к какому-либо целостному состоянию не требуются.

- Вследствие того, что *RDD* является *immutable*, система оптимизирует работу медленных узлов (*stragglers*), запуская копии затратных процессов (*tasks*) на более быстрых узлах. Данный механизм не требует специальной синхронизации и обеспечивает сбалансированность нагрузки.

- Планировщик *Apache Spark* при планировании ресурсоемких операций над *RDD* распределяет их по узлам с учетом размещения (*locality*) данных, что в некоторых случаях позволяет уменьшить накладные расходы на операции чтения/записи.

Приведенные выше характеристики делают *Apache Spark* привлекательным инструментом для анализа данных.

2.4 Анализ *Apache Spark MLlib*

MLlib — библиотека *Spark* для машинного обучения.

Данная библиотека была создана для того, чтобы сделать машинное обучение простым и масштабируемым. Она включает в себя распространенные алгоритмы обучения, решающие задачи классификации, регрессии, кластеризации, коллаборативной фильтрации, понижения размерности и другие.

Библиотека *MLlib* состоит из **2**-х модулей:

1. **spark.mllib**[1]

   Содержит *API* для работы с *RDD*

2. **spark.ml**[2]

   Содержит высокоуровневый *API* для работы с *DataFrame*[3] и создания *ML-конвейеров*[4].

В данной работе используется spark.ml по следующим причинам:

1. Разработчики рекомендуют работать именно с этим модулем, ссылаясь на его гибкость и большой спектр предоставляемых инструментов

2. *spark.ml* активно развивается (появился сильно позже *spark.mllib* и включает в себя почти все его возможности)

---

[2]Подробная информация spark.apache.org/docs/latest/ml-guide

[1]Подробная информация spark.apache.org/docs/latest/mllib-guide

[3]Подробная информация spark.apache.org/docs/latest/sql-programming-guide#dataframes

[4]Подробная информация spark.apache.org/docs/latest/ml-guide.html#pipeline

[5]Подробнее о концепции *transformer-estimator* будет рассказано в следующем разделе

3. Алгоритмы частичного обучения хорошо соответствуют концепции *transformer-estimator*[5], которая лежит в основе *spark.ml*

4. Более простая интеграция по сравнению с *spark.mllib*

Обзор *spark.ml*

Основные концепции, на которые опирается модуль *spark.ml*:

- **Transformer**: Алгоритм, преобразующий по некоторому правилу один *Dataframe* в другой.
- **Estimator**: Алгоритм, получающий на вход *Dataframe* и конструирующий на его основе *Transformer*
- **Pipeline**: Структура, связывающая в один поток несколько *Transformer* и *Estimator*

Пусть имеется задача классификации документов. Для демонстрации работы *Pipeline* одно из её возможных решений представлено на схеме:

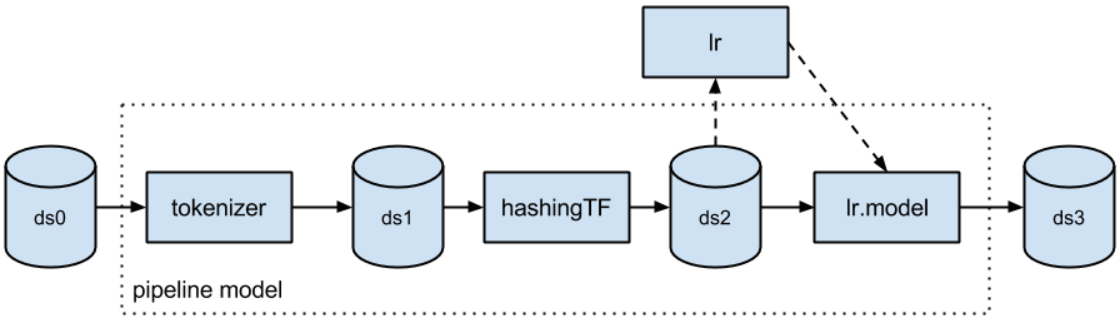


Рис. 3: Схема работы *Pipeline* (с ресурса *databricks.com*)

Шаги работы алгоритма:

1. входное множество документов, представленное датафреймом *ds0*, токенизируется с помощью *tokenizer*, который является *Transformer*

2. результат токенизации *ds1* поступает на вход *Transformer hashingTF*, который извлекает из него признаки *TF-IDF*

3. полученные признаки подаются на вход *Estimator lr*, который тренирует модель логистической регрессии *lr.model*, представляющую из себя *Transformer*

4. на выходе получается модель *Pipeline*, которая является *Transformer* и может применяться для предсказания классов, преобразования исходных данных и получения той или иной статистики по ним.

2.5 Структура классов модуля *spark.ml*

Будет рассмотрена часть модуля, затронутая в рамках данной курсовой, а именно: отвечающая за задачи классификации.

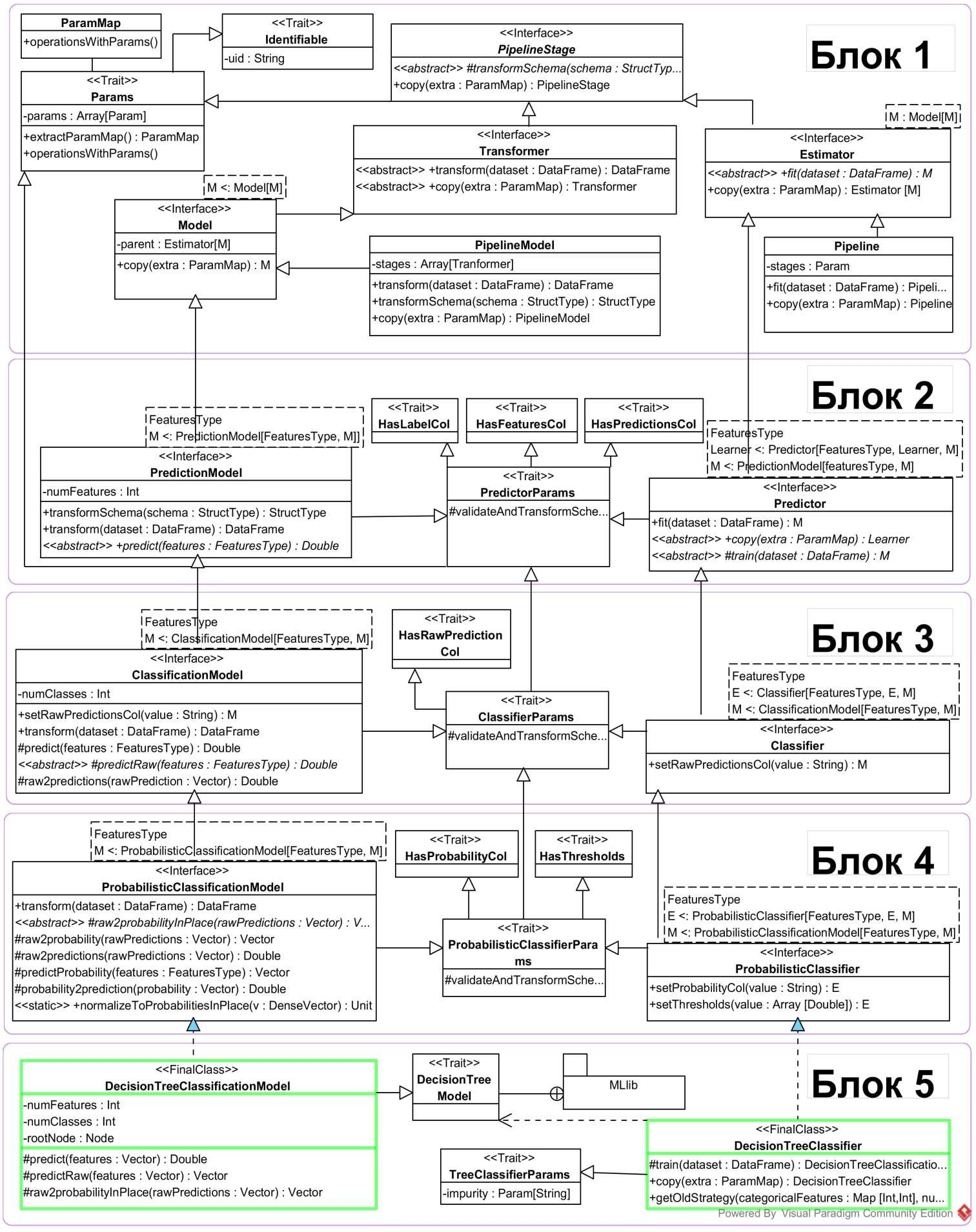


Рис. 4: Диаграмма классов *spark.ml*

Диаграмму по иерархии можно разбить на 5 блоков:

1. базовые интерфейсы конвейера
2. интерфейсы машинного обучения
3. интерфейсы классификаторов
4. интерфейсы для вероятностных классификаторов
5. реализация

Далее приводится краткий обзор классов каждого блока.

Блок 1 : базовые интерфейсы конвейера

В этом блоке определены основные интерфейсы *spark.ml*

- **PipelineStage** — этап конвейера, должен поддерживать метод для трансформации схемы данных во внутреннее представление. Каждый *PipelineStage* - это либо *Transformer*, либо *Estimator*.
- **Estimator** — этап конвейера, создающий *Transformer* на основе переданных в него данных.
- **Transformer** — этап конвейера, преобразующий полученные входные данные в выходные по некоторому правилу.
- **Model** — шаблон с параметром "M - Тип Модели"для *Transformer*. Добавляет в *Transformer* ссылку на породившего его *Estimator*.
- **Pipeline** — конвейер. Создает модель конвейера на основе установленных параметров и определенных пользователем этапов. Этапы хранятся в классе *stages*.
- **PipelineModel** — модель конвейера

Все *Transformers* идут в паре с их порождающими *Estimators*.

Блок 2 : интерфейсы машинного обучения

В этом блоке определены интерфейсы для классов, реализующих алгоритмы машинного обучения (МО).

- **PredictorParams** — трейт для работы с параметрами классов МО.
- **Predictor** — шаблонный интерфейс базового класса-estimator для генерации моделей-transformers МО.
- **PredictionModel** — шаблонный интерфейс модели-transformer'а МО

Данные интерфейсы содержат абстрактные методы *(copy, train,predict)*, которые необхожимо реализовать в конкретных классах МО.

Блок 3 : интерфейсы классификаторов

- **ClassifierParams** — трейт для работы с параметрами классификаторов.
- **Classifier** — шаблонный интерфейс базового класса-estimator, реализующего алгоритм классификации, для генерации моделей-transformers. Представляет из себя обертку над *Predictor*, расширенную параметрами классификации.
- **ClassificationMode** — интерфейс модели-transformer'а МО

В *ClassificationModel* добавляется абстрактный метод *predictRaw()*, с помощью которого реализуются все оставшиеся абстрактные методы базового класса.

Блок 4 : интерфейсы для вероятностных классификаторов

- **ProbabilisticClassifierParams** — трейт для работы с параметрами классификатора.
- **ProbabilisticClassifier** — шаблонный интерфейс базового класса-estimator для генерации моделей-transformers. Добавляет setter'ы для дополнительных параметров (граница принятия и т.п.)
- **ProbabilisticClassifierModel** — интерфейс вероятностного классификатора. Добавляет методы ддля работы с вероятностями предсказаний. Все они используют абстрактные методы *predictRaw()* и *raw2probabilitiesInPlace()*.

Блок 5 : реализация. Пример.

Пример реализации решающих деревьев.

Как видно из диаграммы, решающие деревья используют реализацию из *spark.mllib*.

Также можно видеть, что все классы объявлены как *final*.

2.6 Обзор метрик оценки качества для тестирования

Обозначения:

- TP (*TruePositive*) - количество верно отнесенных к классу 1 точек
- FP (*FalsePositive*) - количество неверно отнесенных к классу 1 точек
- TN (*TrueNegative*) - количество верно отнесенных к классу -1 точек
- FN (*FalseNegative*) - количество неверно отнесенных к классу -1 точек

В библиотеке *MLlib* реализованы следующие метрики для оценки результатов работы алгоритмов:

1. Точность (*precision*) — доля верно размеченных точек из отнесенных к классу 1

$$precision = \frac{TP}{TP + FP}$$

2. Полнота (*recall*) — доля верно отнесенных к классу 1 точек из всех точек класса 1

$$recall = \frac{TP}{TP + FN}$$

3. F-мера (*f-measure*) — среднее гармоническое полноты и точности

$$fmeasure = 2 * \frac{precision \cdot recall}{precision + recall}$$

4. AUC (*AreaUnderCurve*) — площадь под ROC-кривой[1]

$$AUC = \int_0^1 \frac{TP}{TP + FP} d(\frac{TP}{P})$$

2.7 Вывод

Существует большое количество алгоритмов частичного обучения, каждый из которых опирается на те или иные предположения о природе данных. При выполнении этих предположений утверждается, что качество работы будет выше по сравнению с обучением с учителем.

Фреймворк *Apache Spark* позволяет эффективно реализовать алгоритмы частичного обучения, которые в полной мере удовлетворяют интерфейсу активно развивающейся библиотеки *MLLib*.

Также в данной библиотеке имеются реализации метрик для оценки качества работы алгоритмов машинного обучения.

Исходя из этого можно сделать вывод, что выбранный инструментарий является достаточным для достижения поставленных целей.

---

[1]Подробная информация en.wikipedia.org/wiki/Receiver_operating_characteristic

3 Исследование и построение решения задачи

В данной главе описан процесс исследования и его результаты на каждом этапе.

В частности, приводится обоснование выбора алгоритмов для реализации в рамках данной работы.

Далее на основе выбора представлен объектно-ориентированный анализ алгоритмов и определена структура соответствующих классов.

Далее описаны возможности фреймворка *Apache Spark*, использованные для оптимизации работы алгоритмов.

На следующем этапе приведено описание природы и свойств данных, которые использовались для тестирования и анализа реализованных алгоритмов.

В последней части описаны результаты тестирования, использованные параметры и проведен анализ выбранных алгоритмов на основе выполненной работы.

3.1 Выбор алгоритмов

Для реализации на *Apache Spark* после проведения анализа были выбраны следующие алгоритмы:

1. **Self-Training**
2. **Co-Training**
3. **EM-алгоритм**

Данные алгоритмы были выбраны по причине простоты, универсальности, схожих принципов работы (каждый из них *wrapper*) и широкой распространенности в среде data scientist'ов.

3.2 ООП анализ и декомпозиция

В ходе анализа было автором принято решение разделить логику работы выбранных алгоритмов частичного обучения на следующие составляющие:

Управление и работа с данными

Причина: алгоритму необходим доступ к размеченным и неразмеченным данным в процессе обучения и проведение специальных операций над ними.

Пример: извлечение признаков по индексам в *CoTraining*
Решение: определить ответственный за работу с данными класс **SSVData**

Управление базовыми классификаторами

Причина: необходимость управления работой базовых классификаторов.

Проблема: необходимые методы классификаторов объявлены как *protected*, а сами классификаторы — как *final*-классы. Единственный способ доступа — через конвейер, поскольку классификаторы являются наследниками *PipelineStage* (см. приведенную выше диаграмму классов)

Решение: осуществлять управление посредством класса-обертки над конвейером **PipelineForSSVLearning**

Проблема: выбор эффективного способа хранения данных.

Решение: хранить данные в **DataFrame**, поскольку это – входной формат конвейера. Также часто проводятся операции исключения/пересечения нескольких датафреймов и если бы данные хранились в **RDD**, необходимы были бы дополнительные преобразования к датафреймам, что является дополнительным накладным расходом ресурсов.

Основная логика

Проблема: необходимо дать пользователю возможность передать классификатору *неразмеченные* данные для обучения, поскольку по основному потоку конвейера передаются только *размеченные*.

Решение: определить ответственный за хранение и доступ из внешнего кода к данным трейт **SSVClassifier**, который подмешивается в основной класс алгоритма.

Проблема: необходима поддержка всех видов базовых классификаторов с учетом их инкапсуляции и статической типизации языка *Scala*.

Решение: определить классы алгоритмов частичного обучения шаблонными и использовать механизм границ типов *Scala*

Проблема: необходима поддержка интерфейса классификаторов *spark.ml* для возможности использования алгоритмов в конвейере.

Решение: сделать основные классы наследниками от *Serializable* и *ProbabilisticClassifier*.

Проблема: необходимо дать пользователю возможность передать классификатору *неразмеченные* данные для обучения, поскольку по основному потоку конвейера передаются только *размеченные*.

Решение: определить ответственный за хранение и доступ из внешнего кода к данным трейт **SSVClassifier**

3.3 Особенности реализации

В ходе написания кода автор столкнулся с некоторыми особенностями разработки под *Apache Spark*. Далее будут две наиболее важные из них.

Большие временные затраты на чтение/запись данных

Это проявилось в том, что время итерации алгоритмов росло почти экпоненциально. Как оказалось, проблема была в следующем: операции с *Dataframe* и *RDD* (например, *join, except*) по умолчанию каждый раз удваивают число разбиений данных на диске (*partitions*), что приводит к вдвое бóльшим затратам на операции чтения[1].

Также проблема была связана с тем, что *Spark* по умолчанию не кэширует данные в оперативной памяти (в целях экономии памяти). Кэширование позволяет проводить операции с данными в разы быстрее, но требует больше оперативной памяти по сравнению с хранением на жестком диске[2].

Так как выбранные датасеты целиком помещаются в оперативную память, было принято решение кэшировать их каждый раз, если они используются от 2 и более раз. В частности, неразмеченные и размеченные данные в ходе обучения классификаторов кэшируются.

Дополнительно фиксируется число разбиений этих данных на дисках. Эмпирическим путём установлено, что наибольший выигрыш в эффективности получается при количестве разбиений, равном 4-м.

Организация доступа к данным

В *Spark* нет возможности обращаться к записям *Dataframe* или *RDD* по индексу. Для этого необходимо преобразовывать их в массивы, что весьма затратно по времени. Поэтому было принято решение отказаться от индексации и использовать вместо такие операции, как *filter* и *map*, создавая и работая с новыми коллекциями (вместо извлечения нужных полей по индексу происходит фильтрация и отбрасывыние неподходящих).

Важно, что в некоторых случаях *Spark* перераспределяет данные по диску (*shuffling*), что приводит к накладным расходам[3]. Принятое на предыдущем шаге решение отказаться от индексации позволяет частично обойти эту проблему (так как работа по индексам с распределенными данными вызывала бы их *reshuffling*, перераспределение).

Кэширование данных использует механизм сериализации. В официальной документации указано, что наиболее эффективным сериализатором является *KryoSerializer* [4], который и используется в данной работе.

---

[1]Подробнее stackoverflow.com/questions/31659404

[2]Подробнее sujee.net/2015/01/22/understanding-spark-caching

[3]Подробнее blog.cloudera.com/blog/2015/03/how-to-tune-your-apache-spark-jobs-part-1/

[4]Подробнее spark.apache.org/docs/latest/tuning

3.4 Наборы данных для тестирования

За основу были взяты 4 набора данных, использованных в работе *O. Chapelle* и *B. Schlkopf "Semi-Supervised Learning"*[5] и 1 набор из стороннего источника[4].

Описания наборов:

1. **Digit1**

   Данные являются сгенерированными изображениями цифры "1" размером $16x16$. Классы 1 и -1 отвечают за наклон цифры — влево либо вправо соответственно. Далее был добавлен гауссов шум.

2. **USPS**

   Данные представляют собой 150 картинок для каждой из 10 цифр из открытого набора данных *USPS*. Цифры "2" и "5" отнесены к классу 1, остальные к -1.

3. **g241c**

   Данные получены из иножества изотропных гауссиан с единичной дисперсией. Классы точек 1 и -1 различаются центрами скоплений порождающих гауссиан. Центры гауссиан находятся на расстоянии 2.5 друг от друга ($||\mu_1 - \mu_2|| = 2.5$)

4. **g241d**

   Данные получены из 2-х изотропных гауссиан с единичной дисперсией, затем стандартизованы по каждому измерению. Центры гауссиан находятся на расстоянии 2.5 друг от друга ($||\mu_1 - \mu_2|| = 2.5$)

5. **mailSpam**

   Данные представляют собой информацию из спам-фильтра за определенный промежуток времени. Точки из класса 1 — не спам, из -1 — спам.

   Данные предоставлены компанией Mail.ru на одном из соревнований по машинному обучению. Смысл признаков скрыт, о распределении точек ничего не известно.

	кол-во точек	**кол-во признаков**	**баланс**	**cluster a.**	**manifold a.**
Digit1	1500	241	✓	×	✓
USPS	1500	241	×	✓	✓
g241c	1500	241	✓	✓	×
g241d	1500	241	×	×	×
mailSpam	17417	102	?	?	?

Таблица 1: Свойства датасетов

[4]Материалы с дисциплины "Алгоритмы обработки больших объемов данных" курса "Техносфера" (sphere.mail.ru)

3.5 Выбор метрик и методики анализа

Для всестороннего анализа работы алгоритмов было принято решение использовать все доступные метрики и дополнительно такую характеристику, как время работы в секундах.

Имеется 5 степеней свободы для анализа результатов:

1. 5 датасетов
2. 5 алгоритмов с настраиваемыми параметрами
3. 5 метрик
4. процент доступных данных
5. базовые классификаторы

Были приняты следующие решения:

- Наряду с реализованными протестировать алгоритм обучения с учителем **Random Forest**[1]

  Цель — сравнить реализованные алгоритмы частичного обучения и с алгоритмом обучения с учителем

- Подобрать и фиксировать наилучшие параметры алгоритмов для 1-го набора данных

  Полный анализ получился бы очень обширным, поэтому параметры алгоритмов фиксированы. Разумеется, на практике необходимо подбирать параметры для каждого датасета.

- Обеспечить одинаковые условия работы алгоритмов и возможность их кросс-валидации

  Для этого был определен класс **SSVTester**, о котором будет написано ниже.

- Проанализировать зависимость качества работы от количества размеченных точек по сравнению с базовым классификатором на использованных при обучении неразмеченных точках и резервных (неиспользованных).

  Для этого для каждого датасета перебирается процент размеченных точек на множестве $\{0.005, 0.0075, 0.01, 0.015, 0.025, 0.05, 0.1, 0.15, 0.2, 0.4\}$, берётся 10 случайных разбиений на размеченную, неразмеченную и резервную выборки, затем вычисляются средние значения метрики *F1* и доверительные интервалы на результатах классификации неразмеченный и резервной выборок. По полученным сериям строятся графики.

  Данная процедура позволяет определить, насколько лучше алгоритмы частичного обучения справляются с задачей классификации видимых ранее и новых данных при малом количестве размеченных точек, чем алгоритмы обучения с учителем.

[1] Подробно ознакомиться с принципом работы можно в статье *Breiman L. "Random forests"* [3]

- Определить зависимость качества алгоритмов от максимального количества совершаемых итераций

  Данный параметр перебирается на отрезке $[3, 10]$ с шагом 1 при 3% размеченных данных на первых 4-х датасетах.

- Определить зависимость качества *SelfTraining* и *CoTraining* от границы принятия данных (уверенности классификации)

  Данный параметр перебирается на отрезке $[0.60, 1.0]$ с шагом 0.1 при 3% размеченных данных на первых 4-х датасетах.

- Сравнить время работы алгоритмов

  Для этого вычисляется время работы в среднем по каждому алгоритму и датасету.

- Получить среднюю оценку качества по метрикам *AUC* и *F1-measure* для каждого датасета

  Для этого составляется таблица для каждого датасета и алгоритма с лучшими результатами по указанным метрикам. метрике.

3.6 Параметры алгоритмов

В ходе экспериментов использовались следующие параметры алгоритмов.
Количество деревьев **RandomForest**: 128.

	базовая м.	**max. итераций**	**гр. принятия**
Self-Training	RF	3	0.95
Co-Training	RF	3	0.90
EM	RF	10	–

Таблица 2: Параметры алгоритмов

Разделение по признакам в алгоритме **Co-Training** случайное, на 2 равномощных множества.

Окрестность сходимости **EM**: 3%

Параметры были подобраны эмпирически.

3.7 Результаты работы

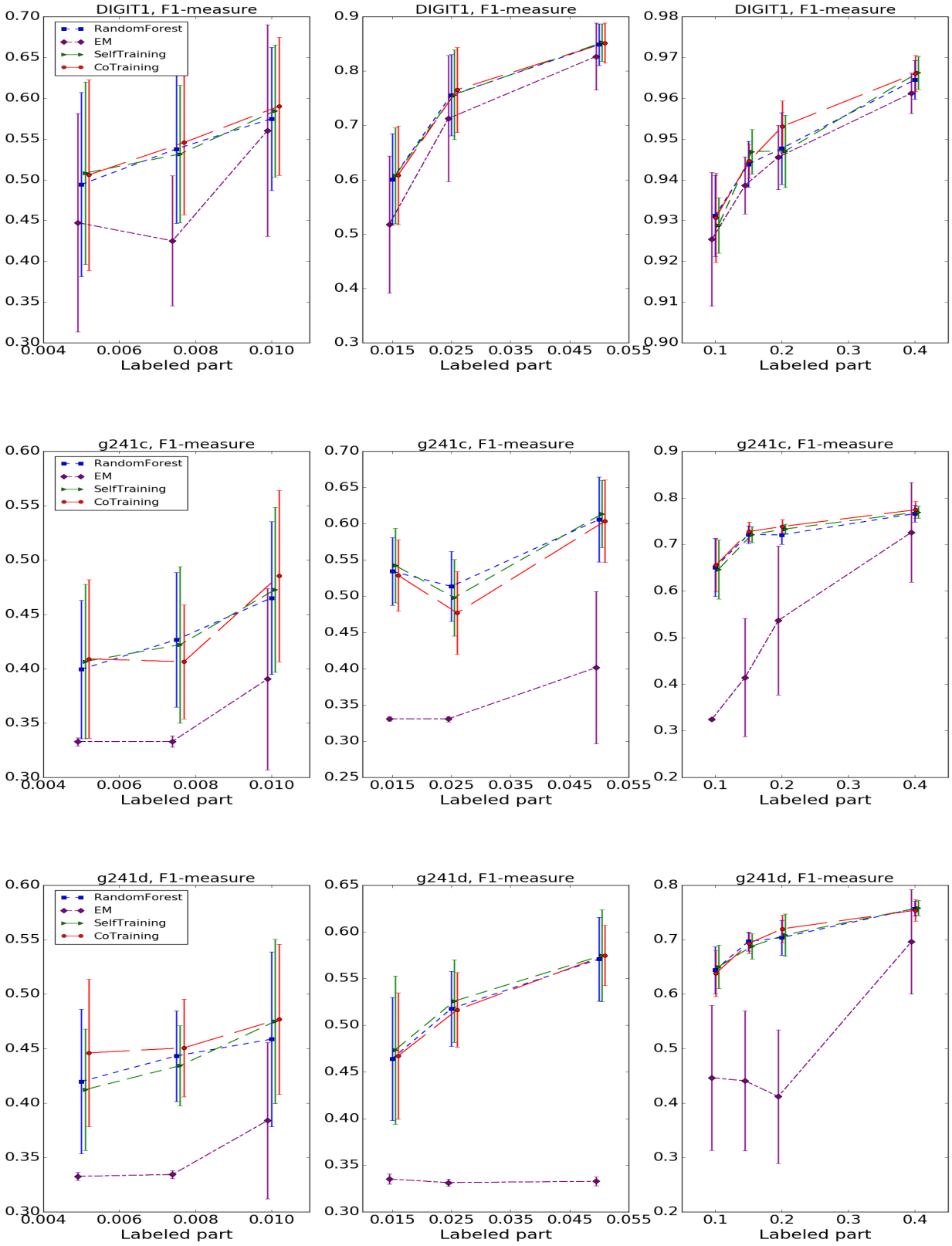


Рис. 5: Зависимость AUC и F1-measure от кол-ва разм. данных

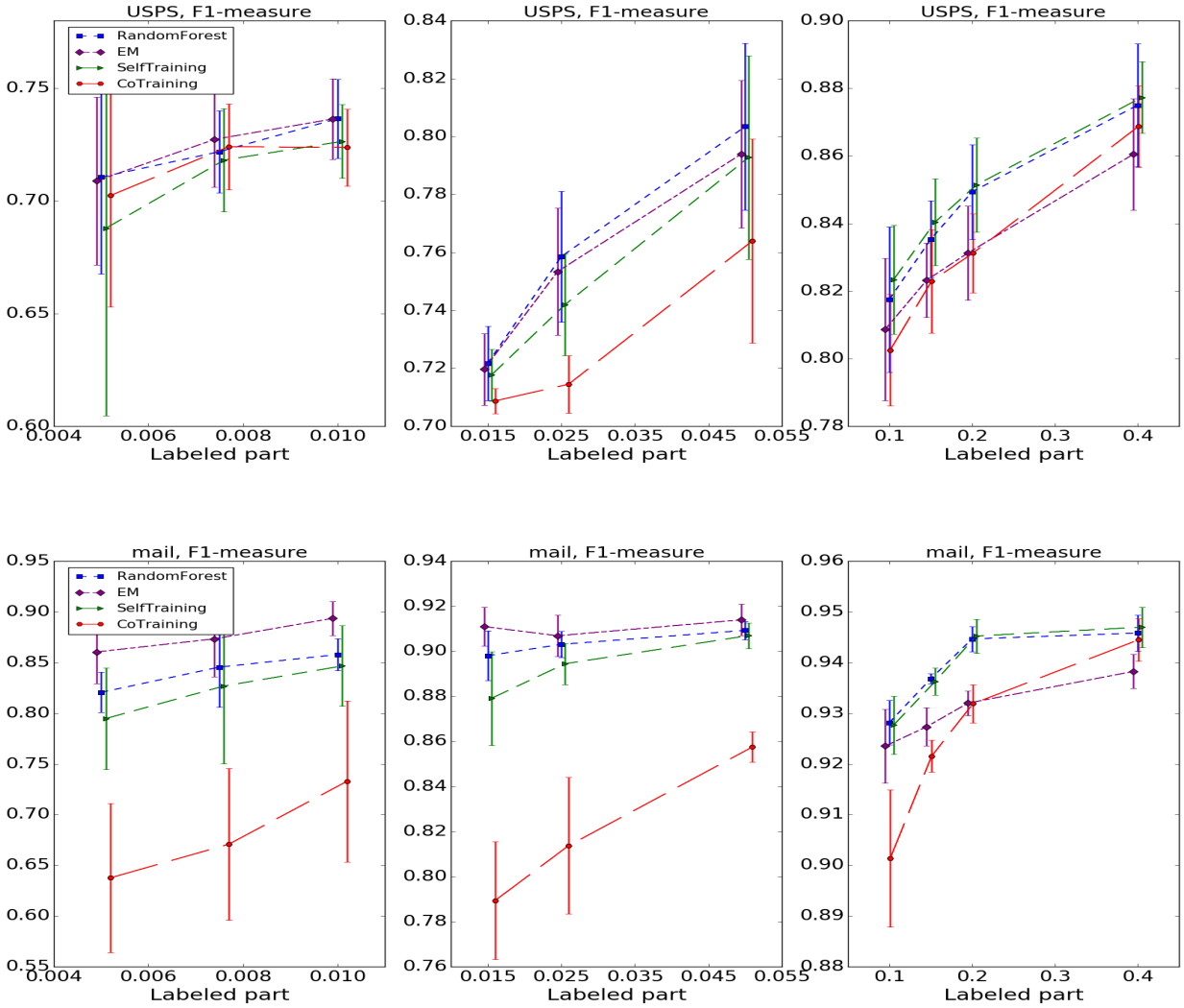


Рис. 6: Зависимость AUC и F1-measure от кол-ва разм. данных

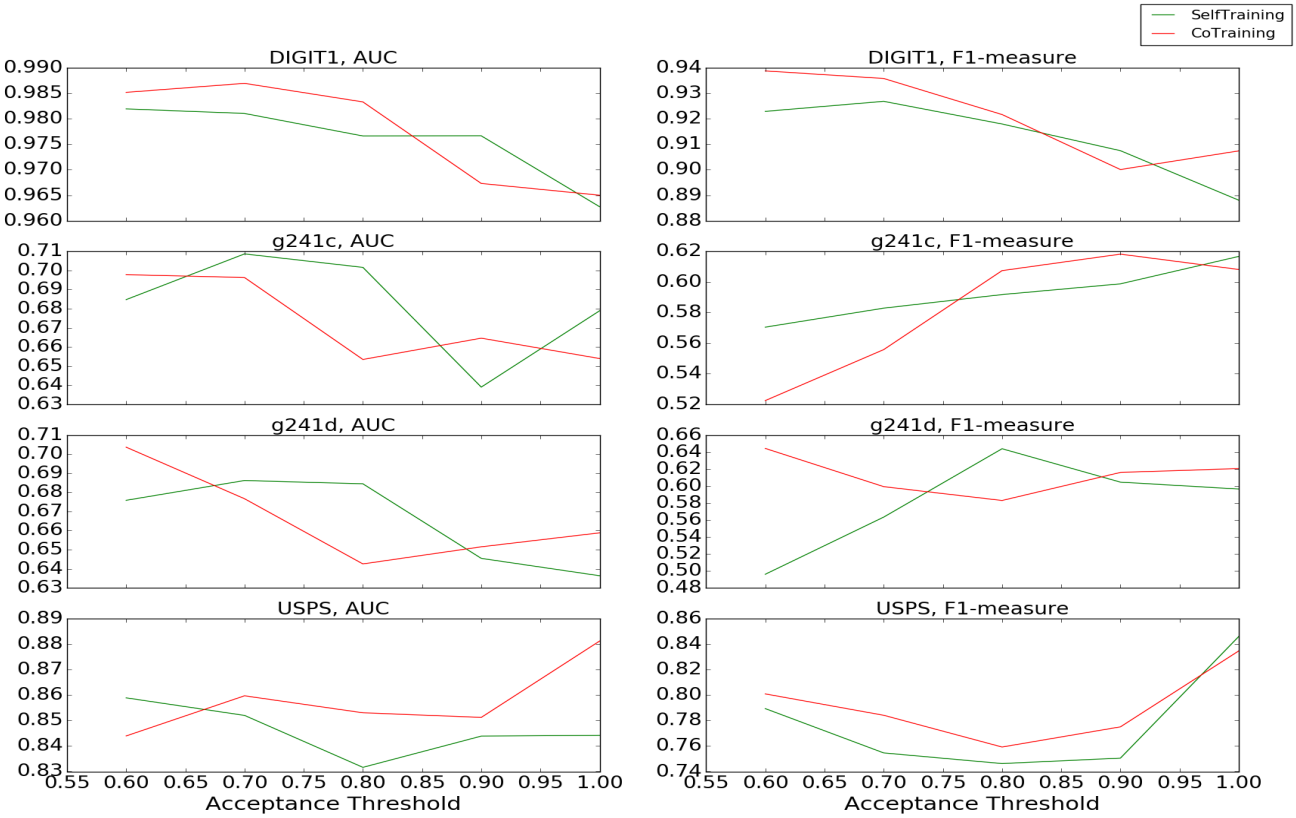


Рис. 7: Зависимость AUC и F1-measure от порога принятия

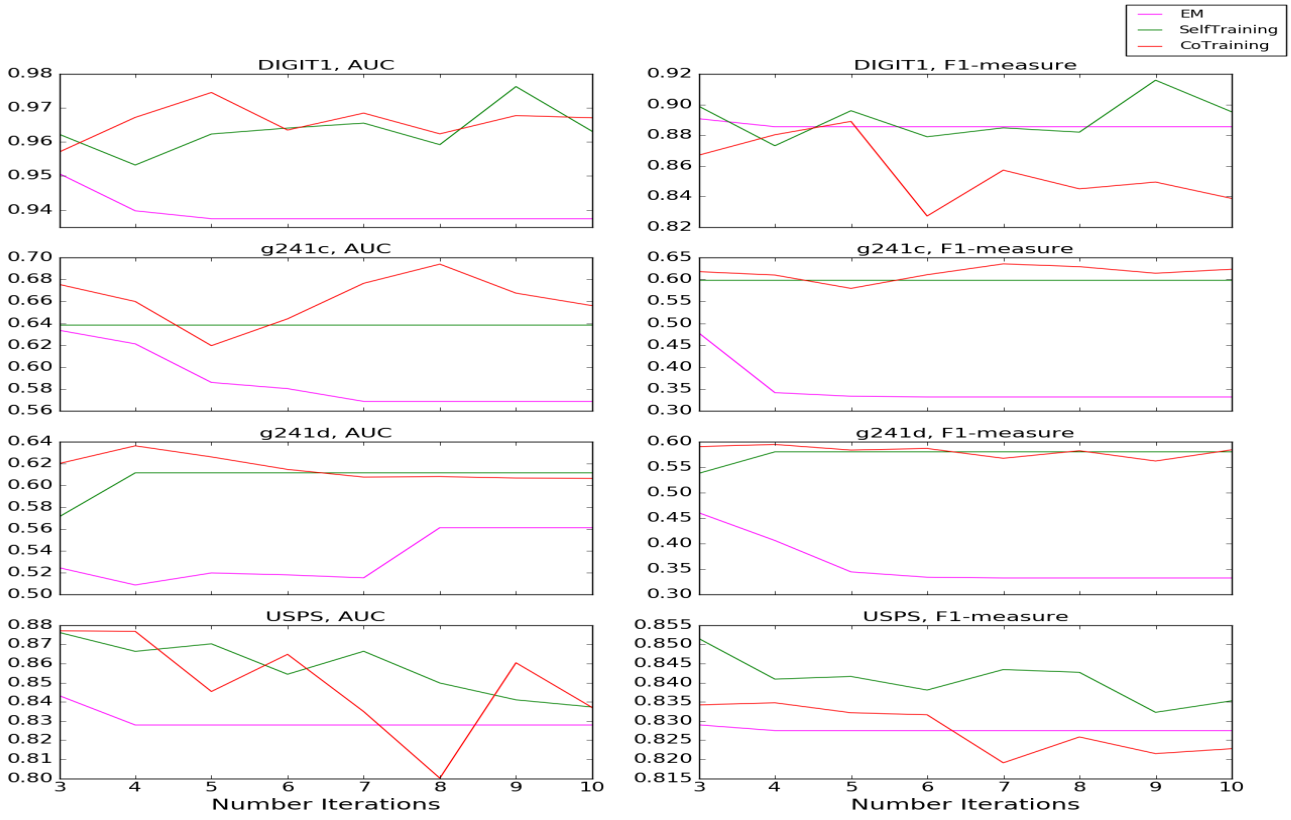


Рис. 8: Зависимость AUC и F1-measure от макс. кол-ва итераций

	Random Forest	*Self-Training*	*Co-Training*	*EM*
DIGIT1	0.760 ± 0.052	0.763 ± 0.050	**0.766** ± 0.052	0.726 ± 0.068
g241c	0.580 ± 0.047	**0.583** ± 0.047	0.581 ± 0.048	0.412 ± 0.060
g241d	0.567 ± 0.045	0.570 ± 0.045	**0.574** ± 0.043	0.405 ± 0.057
USPS	**0.783** ± 0.020	0.778 ± 0.024	0.767 ± 0.019	0.776 ± 0.024
mailSpam	0.899 ± 0.010	0.890 ± 0.021	0.820 ± 0.032	**0.908** ± 0.012

Таблица 3: Средняя мера F1 по датасетам

	Random Forest	**Self-Training**	**Co-Training**	**EM**
DIGIT1	**12.29**	46.45	117.34	140.17
g241c	**11.65**	27.12	53.49	131.23
g241d	**12.49**	28.79	56.30	145.91
USPS	**17.57**	101.12	214.51	136.41
mailSpam	**12.61**	68.08	134.34	137.28

Таблица 4: Среднее время обучения по датасетам

3.8 Вывод

Из результатов проведенного исследования можно сделать следующие выводы:

- Результат очень сильно зависит от начальных размеченных данных.

  Если данные репрезентативны, то результат алгоритмов частичного обучения будет лучше результата базовой модели, иначе — сильно хуже.

- Ошибка классификации многократно множится при неправильном выборе границы принятия.

  На графиках можно наблюдать, как при "проседании"базовой модели сильно падают модели частичного обучения.

- *EM*-алгоритм работает непредсказуемо на некоторых данных (по предположению автора, "бросается"от одного экстремума оценки правдоподобия к другому)

Зависимость от предположений		
Clust.	Manif.	Вывод
✓	✓	• Наравне с базовой моделью • Сильно зависит от инициализации — ошибка многократно множится, из-за чего лучше брать максимальную границу принятия и небольшое число итераций
×	✓	• Незначительно превосходит базовую модель • Границу принятия не следует делать слишком высокой (поскольку точки на меньшей размерности имеют примерно одинаковые уверенности, а цель — взять их верхний слой) • Функция качества от количества итераций имеет единственный экстремум (после которого начинается переобучение)
✓	×	• *EM* работает плохо, остальные несильно уступают базовой моделью • Границу принятия следует брать небольшой (поскольку все соседние от размеченной точки находятся в одном кластере, в них классификатор примерно одинаково уверен, в остальных же точках — заметно меньше)
×	×	• *EM* работает плохо, остальные несильно превосходят базовой моделью • Проблема усиления ошибки не так явно выражена, поскольку природа данны

4 Описание практической части

В данной главе приведено описание реализации, а именно: диаграммы классов и настраиваемые параметры.

Также выведены оценки сложности и потребляемой памяти.

4.1 UML диаграммы классов

Классы реализованных алгоритмов имеют схожий вид и интерфейс: все они являются шаблонными, реализуют *ProbabilisticClassifier*, используют *PipelineForSSVLearning* и включают в себя трейт *SSVClassifier*.

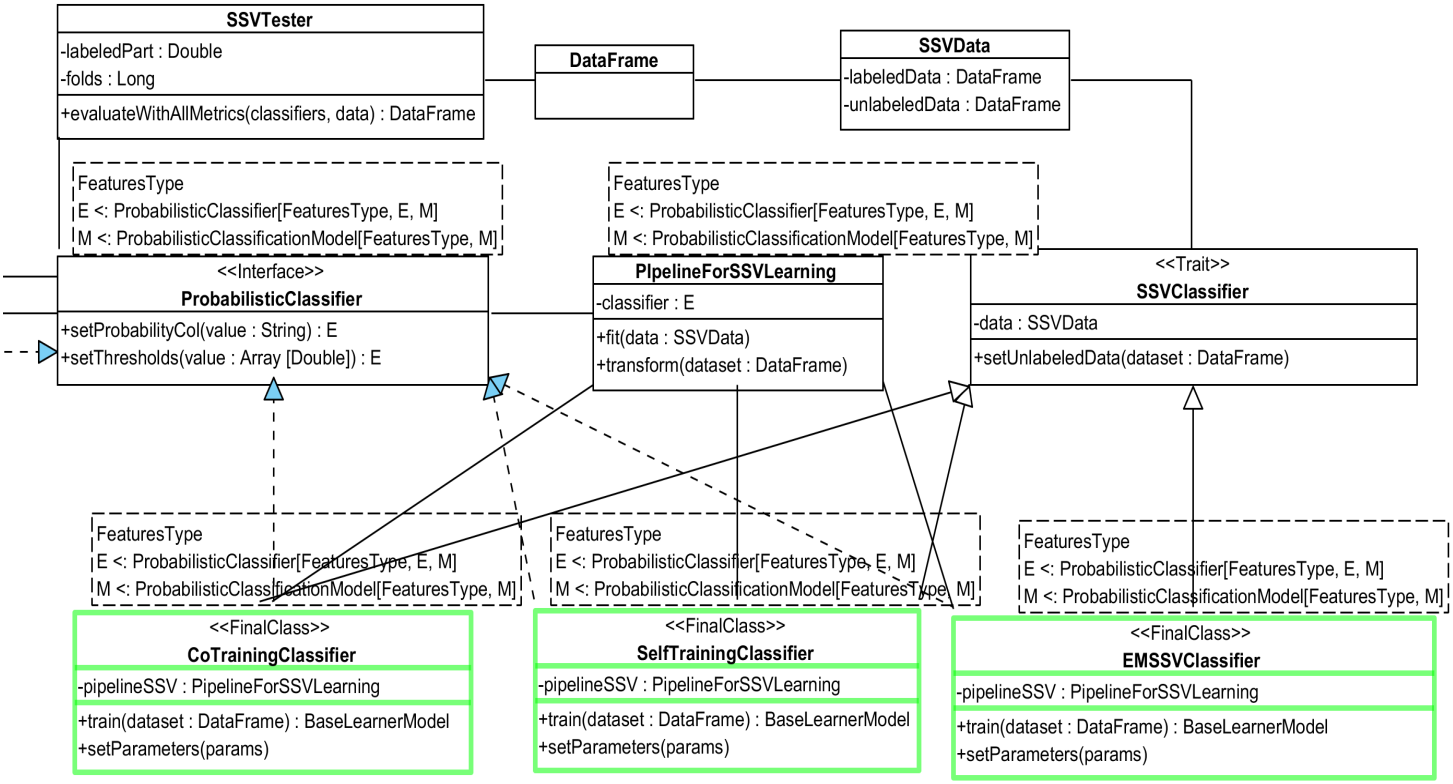


Рис. 9: Диаграмма классов *semi-supervised learning*

4.2 Настраиваемые параметры алгоритмов

Тестер SSVTester:

- *folds* — количество разбиений данных
- *labeledPart* — доля размеченных данных ($(0; 1]$)
- Данные, с которыми предстоит работать

  Передаются в формате *DataFrame* в метод *setData(. . .)*, далее преобразовываются во внутренний формат.

Self-Training и Co-Training:

- *baseClassifier* — базовая(-ые) модель(-ли)
- *finalClassifier* — итоговая модель (для *Co-Training*, в *Self-Training* базовая модель является и итоговой)
- *countIterations* — максимальное число итераций
- *thresholdTrust* — граница принятия (достаточная уверенность классификатора, чтобы принять точку в размеченный набор)
- *verbose* — вывод отладочных данных (*True* или *False*)

EM:

- *baseClassifier* — базовая (она же и итоговая) модель
- *countIterations* — максимальное число итераций
- *minResidualPercent* — окрестность сходимости (отношение различно классифицированых точек к общему числу точек на последовательных итерациях)
- *verbose* — вывод отладочных данных (*True* или *False*)

4.3 Оценка сложности и потребление памяти

Пусть L и U — количество размеченных и неразмеченных точек соответственно, k — максимальное количество итераций.

Каждый из алгоритмов совершает максимум k итераций, на каждой из которых происходит обучение базового классификатора (или 2-х в случае *Co-Training*) и предсказание неразмеченных данных, а также выделение новых точек (для *Self-Training* и *Co-Training* , за линейное от U время) либо подсчет невязки с предыдущей итерацией для определения сходимости (для *EM*, за линейное от $U + L$ время).

Важно заметить, что оценки являются оптимальными.

Память тратится на то, чтобы хранить проиндексированные размеченные и неразмеченные данных (либо еще и исходные для *Co-Training*). Итого расход линеен по $U + L$.

Алгоритм	**Время**	**Память**
Self-Training	$O(k \cdot (T^{train}_{base\ learner} + T^{fit}_{base\ learner} + U))$	$O(U + L)$
Co-Training	$O(k \cdot (T^{train}_{base\ learner} + T^{fit}_{base\ learner} + U))$	$O(U + L)$
EM	$O(k \cdot (T^{train}_{base\ learner} + T^{fit}_{base\ learner} + U + L))$	$O(U + L)$

Таблица 5: Оценки алгоритмов

Ссылки на код

В работе была использована версия *Apache Spark* 1.6.1

Код выложен в открытый репозиторий на https://github.com/hbq1/SSL_last.

4.4 Вывод

Описанная реализация полностью удовлетворяет интерфейсу библиотеки *MLLib*, а также по максимуму использует её возможности.

Реализация поддерживает настройку всех возможных параметров алгоритмов, что позволяет пользователю адаптировать алгоритм для решения конкретной задачи.

Оценки скорости и потребляемой памяти являются оптимальными.

Заключение

В рамках курсовой работы были получены следующие результаты:

1. Проведен анализ существующих алгоритмов частичного обучения
   - Составлен подробный обзор
2. Написана реализация алгоритмов *Self-Training*, *Co-Training*, *EM* для случая бинарной классификации
   - Составлен обзор фреймворка *Apache Spark*
   - Подробно рассмотрена структура *MLlib*
   - Спроектирована полностью удовлетворяющая интерфейсу *MLlib* структура классов выбранных алгоримов
3. Получены и проанализированы результаты работы реализованных алгоритмов на 5-ти наборах данных
   - Реализован универсальный тестер для задач бинарной классификации
   - Проведен сравнительный анализ с базовыми алгоритмами
   - Проведены тесты на зависимость качества от задаваемых параметров

Список литературы

[1] *A. Criminisi J. Shotton E. K.* Decision forests for classification, regression, density estimation, manifold learning and semi-supervised learning.: Tech. Rep. MSR-TR-2011-114: Microsoft Research, 2011. — Oct.

[2] *Blum A., Mitchell T.* Combining labeled and unlabeled data with co-training // Proceedings of the Eleventh Annual Conference on Computational Learning Theory. — COLT' 98. — New York, NY, USA: ACM, 1998. — Pp. 92–100. http://doi.acm.org/10.1145/279943.279962.

[3] *Breiman L.* Random forests // *Machine Learning.* — 2001. — Vol. 45, no. 1. — Pp. 5–32. http://www.cs.colorado.edu/~grudic/teaching/CSCI5622_2004/RandomForests_ML_Journal.pdf.

[4] *M. Zaharia M. Chowdhury T. D. A. D. J. M. M. M. M. J. F. S. S. I. S.* Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing // Presented as part of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12). — San Jose, CA: USENIX, 2012. — Pp. 15–28. https://www.usenix.org/conference/nsdi12/technical-sessions/presentation/zaharia.

[5] *O. Chapelle B. Schlkopf A. Z.* Semi-Supervised Learning. — 1st edition. — The MIT Press, 2010.

[6] *P. Mallapragada R. Jin A. K. J. Y. L.* Semiboost: Boosting for semi-supervised learning // *IEEE Trans. Pattern Anal. Mach. Intell.* — 2009. — Vol. 31, no. 11. — Pp. 2000–2014. http://dx.doi.org/10.1109/TPAMI.2008.235.

[7] *Saffari A.* Multi-Class Semi-Supervised and Online Boosting: Ph.D. thesis / Graz University of Technology, Faculty of Computer Science. — 2010.

[8] *Vapnik V. N.* Statistical Learning Theory. — Wiley-Interscience, 1998.

[9] *Zhu X.* Semi-supervised learning literature survey: Tech. Rep. 1530: Computer Sciences, University of Wisconsin-Madison, 2005.