Inhaltsverzeichnis

Übung 1: Ein-, Aus- und Weitergabe

1. Öffnen Sie eine SSH-Verbindung zum Rechner `ella`.
2. Starten Sie einen lange (d.h. länger als eine Stunde) laufenden Prozess.
3. Schieben Sie den Prozess in den Hintergrund.
4. Holen Sie den Prozess wieder in den Vordergrund
5. Brechen Sie den Prozess ab
6. Legen Sie in Ihrem Homeverzeichnis das Verzeichnis `My directory` an.
7. Legen Sie nun ein Verzeichnis namens `My sister's directory` an.
8. Löschen Sie beide Verzeichnisse wieder.
9. Legen Sie nun in Ihrem Homeverzeichnis ein Verzeichnis `Vorkurs` an.
10. Erzeugen Sie in dem neu erstellten Verzeichnis eine Datei `poetik.txt`, in der Sie *alle* Dateien aus dem Verzeichnis `/resources/corpora/monolingual/raw/projekt_gutenberg/werke/aristote/poetik/cleaned` aneinanderhängen (konkatenieren).
11. Wir wollen nun herausfinden, welche Wörter die häufigsten in diesem Korpus sind.

    Tipp: Lesen Sie die man-page zu den erwähnten Tools.

    a) Der offensichtliche Weg zum zählen von Vorkommen in Dateien ist `grep`. Damit können wir z.B. zählen, wie oft das Wort *Epos* vorkommt. Tun Sie das doch einfach mal mit der Datei poetik.txt; zählen Sie die Wörter *Epos*, *der*, *Der* und *Melodik*. Tipp: Der reguläre Ausdruck `\b` matcht Wortgrenzen.

    b) Per default identifiziert `grep` allerdings Zeilen und keine Wörter. Unsere Zählung aus der vorherigen Aufgabe ist also etwas ungenau – Immer wenn das Wort zwei Mal in der gleichen Zeile vorkommt, verzählen wir uns.

    c) `grep` bietet die Option `-o`. Damit werden nicht Zeilen gefunden, sondern jedes einzelne Vorkommen des Musters wird aufgelistet. In Verbindung mit `wc` können wir nun alle Vorkommen der Wörter *Epos*, *der*, *Der* und *Melodik* zählen.
12. Dummerweise ist das recht umständlich auf diese Weise. Für eine komplette Häufigkeitsverteilung bräuchten wir eine Liste aller Wörter – die wir nicht haben. Wir gehen also etwas anders an die Sache heran. Die nächsten Schritte sind die einzelnen Teile einer Pipe, Sie müssen die Ausgabe also nicht speichern. Wenn Sie wissen, wie jeder Schritt geht (und ihn in ihrer Konsolen-History haben), reicht das.

    a) Da wir Wörter besser zählen können, wenn nur eines auf einer Zeile steht, sollten wir erstmal dafür sorgen, dass jedes Wort auf einer eigenen Zeile steht (verwandeln Sie Leerzeichen in Zeilenumbrüche, indem Sie `sed` oder `tr` und ggf. reguläre Ausdrücke verwenden).

b) Würden wir die Wörter jetzt zählen, hätten wir noch Satzzeichen wie Kommas, Punkte, Klammern u.a., die die Ergebnisse verunreinigen. Wir werden also als nächstes die Ausgabe von Schritt 12a um ihre Satzzeichen „erleichtern". Denken Sie an Charakterklassen bei regulären Ausdrücken.

c) Die eigentliche Zählung folgt nun und besteht aus drei Schritten. Wir machen uns dabei die Option `-c` des Kommandos `uniq` zunutze. Damit gibt `uniq` aus, wieviele Zeilen es jeweils zusammengefasst hat. Da `uniq` aber nur untereinanderstehende Zeilen zusammenfasst, sollten wir die Zeilen erst sortieren. Wollen wir die Wörter dann noch nach ihrer Häufigkeit sortieren, sortieren wir die `uniq`-Ausgabe (Schauen Sie auch mal in die man-page von `sort`, um die richtige Sortier-Art festzulegen).

Am Ende kommt eine Pipe raus, die aus ca. fünf Schritten besteht (es gibt mehr als eine mögliche Lösung). Damit Sie vergleichen können, finden Sie die zehn häufigsten Wörter in der Datei `/home/public/vorkurs_ws17/poetik10.txt`.

Zusatzaufgaben

1. Als wir vorhin die Dateien zusammengefasst haben, haben wir das Inhaltsverzeichnis mitgenommen. Es steckt in der Datei `Druckversion_poetik_clean.html`. Wie muss der Befehl aussehen, damit das Inhaltsverzeichnis ausgeschlossen wird? (Benutzen Sie die Pfeiltasten, um vorherige Befehle zu verändern und erneut zu benutzen).

Übung 2: Batch-Verarbeitung

1. Setzen Sie die Variable `FIRSTNAME` auf Ihren Vornamen.
2. Geben Sie den Inhalt der Variablen `FIRSTNAME` aus.
3. Starten Sie nun einen Bash-Subprozess.
4. Geben Sie den Inhalt der Variablen `FIRSTNAME` aus.
5. Beenden Sie den Subprozess.
6. Setzen Sie die Variable `FULLNAME`, indem Sie Ihren Nachnamen an die Variable `FIRSTNAME` anhängen.
7. Schauen Sie sich an, auf was Ihre `$PATH`-Variable gesetzt ist.
8. Ergänzen Sie Ihre `$PATH`-Variable um das Verzeichnis `~/Vorkurs`, das Sie in einer vorherigen Übung angelegt haben. `~/Vorkurs` gehört damit zu Ihrem Suchpfad – ausführbare Programme, die darin liegen, werden dann automatisch gefunden.
9. In dieser Aufgabe sollen Sie nun ein kleines Shell-Skript `overlap.sh` schreiben. Das Skript soll zwei Verzeichnisse als Argumente nehmen. In den beiden Verzeichnissen steht jeweils ein Korpus, der auf verschiedene Dateien verteilt sein kann. Das Skript soll letzten Endes eine Liste der Wörter ausgeben, die in *beiden* Texten vorkommen. Benutzen Sie Ihren Lieblingseditor um das Skript zu schreiben.

   Einige Hinweise:

   - Der erste Schritt ist, die Dateien aus den beiden Verzeichnissen zusammenzufassen. Das haben wir in vorigen Übungen schon gemacht.
   - Danach ist für jeden Text eine Liste der Begriffe zu erstellen, die darin vorkommen. Häufigkeiten sind diesmal nicht von Interesse.
   - Benutzen Sie das Programm `comm`, um den tatsächlichen Overlap ermitteln zu lassen. Wie es genau funktioniert, erfahren Sie auf der zugehörigen man-page.
   - Zwischendurch lässt es sich nicht vermeiden, Dateien zu erzeugen und zu speichern. Tun Sie das *n*icht in Ihrem Homeverzeichnis, sondern in einem temporär erzeugten. Mit `mktemp` kann man temporäre Dateien und Verzeichnisse erzeugen, ohne sich um Dinge wie Namensgleichheit kümmern zu müssen (auf der man-page von mktemp finden sich die Details).
   - Um das Skript auszuführen, müssen Sie es einmal ausführbar machen: `$ chmod u+x overlap.sh`. Danach können Sie `overlap.sh` wie jeden anderen Befehl auch benutzen.

Übung 3: Werkzeuge (AWK und sed)

1. Kopieren Sie die Datei `/home/public/vorkurs_ws17/de_patents.tsv` in Ihr `Vorkurs`-Verzeichnis. Die Datei enthält Metadaten für deutsche Patente in fünf TAB-separierten Spalten (Patentnummer, Datum, Titel, Firma, Erfinder (einzelne Erfinder sind durch „ ,“ getrennt))

   Hinweis: Die meisten dieser Aufgaben lassen sich mit AWK oder mit Unix Tools lösen. Probieren Sie beides aus! Achten Sie darauf, dass AWK standardmäßig Leerzeichen als Spaltentrenner interpretiert, die Datei aber TAB-separiert ist.

2. Lassen Sie sich nur die 1. und 3. Spalte anzeigen.

3. Lassen Sie sich die 5 Spalten in einer anderen Reihenfolge ausgeben.

4. Geben Sie jetzt nur jede 3. Zeile aus.

5. Schreiben Sie für jedes Patent der Firma Siemens nur den Titel des Patents heraus (achten Sie auf Großschreibung!).

6. Ersetzen Sie die Spaltentrenner durch `" ||| "`

7. Schreiben Sie eine `Pipe`, die die einzelnen Erfinder absteigend sortiert nach der Anzahl ihrer Erfindungen ausgibt. Wie heißt die Frau mit den meisten Patenten?

Übung 4: Ressourcen

1. Welche Versionen von WordNet sind bei uns installiert?
2. Welche Korpora sind bei uns installiert, die japanischen Text enthalten?
3. Was müssen Sie beim Zugriff auf die Chinese Treebank beachten?
4. Was müssen Sie bei der Benutzung der Ressourcen i.A. beachten?
5. Was tun Sie, wenn sie festgestellt haben, dass die Angaben im Wiki nicht korrekt sind?

Übung 5: Formate

1. Basics

   a) Kopieren Sie sich die Datei `/home/public/vorkurs_ws18/starwars.xml` in Ihr Vorkurs-Verzeichnis.

   b) Öffnen Sie die Datei mit einem Text-Editor und fügen Sie die Daten von Prinzessin Leia hinzu. Leias Heimatplanet ist Alderaan. Speichern Sie die Datei.

   c) Lesen Sie die XML-Datei nun mit einem DOM-Parser ein. Schreiben Sie nun ein kurzes Python-Skript, das Name und Heimatplanet der Personen ausgibt. Die Ausgabe sollte so aussehen:

   ```
   Luke Skywalker: Tatooine
   Han Solo: Corellia
   Prinzessin Leia: Alderaan
   ```

   d) Ändern Sie das Skript noch einmal, so dass nur Jedis ausgegeben werden, also nur Personen bei denen das tag `<jedi />` definiert ist.

   e) Erstellen Sie ein Dokument `starwars.json` und versuchen Sie, die Daten in `starwars.xml` in JSON-Format zu übertragen. Was ist problematisch?

   f) Verwenden Sie das `json`-Modul im Python-Interpreter, um `starwars.json` einzulesen. Falls die Datei nicht eingelesen werden kann, korrigieren Sie sie.

Zusatzaufgabe Probieren Sie, Aufgaben 1c und 1d mit einem SAX-Parser zu lösen. Was ist einfacher/schwieriger?

2. BNC-Preprocessing

   a) Der British National Corpus ist ein großes Korpus. Wir arbeiten, um die Rechner nicht unnötig zu beanspruchen, daher mit jeweils nur einer Datei. Schauen Sie auf die Uhr und merken Sie sich die letzte Stelle der aktuellen Minutenzahl. Wenn das eine 9 ist, nehmen Sie stattdessen den Buchstaben P. Nun kopieren Sie sich aus `/resources/corpora/monolingual/annotated/bnc/Texts/A/A0/` die Datei, die sich ergibt, wenn sie an A0 die gemerkte Zahl oder den Buchstaben anhängen (sie sollten dann einen Dateinamen wie z.B. A05.xml haben) in Ihr Vorkurs-Verzeichnis.

   b) Wir wollen nun aus dieser Datei den eigentlichen Text des Korpus extrahieren. Erweitern Sie dazu Ihr Python-Skript (Sie brauchen sich eigentlich nur an dem Element `w` zu orientieren, das die Wörter repräsentiert). Entscheiden Sie sich für einen DOM- oder SAX-Parser.

   c) Geben Sie nun statt den Wörtern die Lemmata aus. Die Lemmata stecken im Attribut „hw".

Übung 6: Versionskontrolle

1. Öffnen Sie `https://gitlab.cl.uni-heidelberg.de/opitz/ressourcenvorkurs` in Ihrem Webbrowser.
2. Erzeugen Sie mittels `git clone` eine Kopie des Repositorys (Hinweis: die benötigte URL findet sich auf der Webseite, die sie geöffnet haben).
3. Wechseln Sie in das Verzeichnis, das die Kopie enthält.
4. Öffnen Sie die Datei `database` und fügen Sie irgendwo Ihre Lieblingsfarbe und Ihr Lieblingstier ein.
5. Checken Sie die Änderungen ins Repository ein.
6. Fügen Sie die Änderungen dem Server-Repository zu.
7. Nun wollen wir etwas warten, bis die Änderungen aller anderen im Repository gelandet sind. Unterhalten Sie sich mit Ihrem Nachbarn über Tiere und Farben!
8. Nun aktualisieren Sie Ihre Kopie.
9. Erstellen Sie einen neuen Branch
10. Legen Sie nun eine neue Datei an, in die Sie irgendetwas schreiben. Wenn Ihnen nichts anderes einfällt, schreiben Sie in ein bis zwei Sätzen auf, was Sie letzte Nacht geträumt haben.
11. Fügen Sie die Datei der Versionskontrolle hinzu und committen Sie sie ins Repository.
12. Mergen Sie den erstellten Branch mit `master` und schließen Sie den Branch.
13. Fügen Sie die Änderungen dem Server-Repository zu.

Übung 7: Makefiles

1. Erstellen Sie zunächst eine leere Datei namens `Makefile` in ihrem Vorkurs-Verzeichnis. Kopieren Sie sich dann die Dateien `doc1.tex, doc2.tex` und `lit.bib` aus dem Ordner `/home/public/vorkurs_ws18` in Ihr Vorkurs-Verzeichnis.

2. Mit dem Befehl `pdflatex` können Sie `doc1.tex` in `doc1.pdf` kompilieren. Erstellen Sie eine einfache Makefile, sodass das PDF mit dem Aufruf `:~$ make doc1.pdf` erstellt wird (Hinweis: Sie brauchen nur eine Regel).

3. Definieren Sie jetzt ein Target `all`, das `doc1.pdf` als Abhängigkeit hat. `all` sollte immer am Anfang der Makefile stehen, damit `make` (ohne Angaben) dasselbe tut wie `make all`. Führen sie `make (all)` aus und vergewissern Sie sich, dass das PDF kompiliert wurde. Was passiert, wenn Sie `make all` nochmal ausführen?

4. Fügen Sie eine Regel hinzu, die `doc2.pdf` kompiliert. Fügen Sie `doc2.pdf` dann ebenfalls als Abhängigkeit zum Target `all` hinzu. Führen sie `make all` aus und vergewissern Sie sich, dass das PDF kompiliert wurde.

5. Wenn `make doc2.pdf` aufgerufen wird, warnt `pdflatex`, dass undefinierte Zitationen existieren. `doc2` enthält Zitationen, aber die Bibliographie wurde noch nicht erzeugt. Die bibliographischen Daten stehen in `lit.bib`. Wenn Sie normalerweise `pdflatex` aufrufen, erzeugt es eine Datei `doc2.aux`. Dann muss `bibtex` mit dieser Datei als Argument aufgerufen werden. Erstellen Sie jetzt ein weiteres Target `doc2.bbl`, mit dem Sie die Bibliographie erzeugen. Wenn Sie `:~$ make doc2.bbl` ausführen, sollte eine Bibtex-Datei (endet mit `.bbl`) erzeugt werden. Fügen Sie das target `doc2.bbl` nun auch zu Ihrem Targe `all` als Abhängigkeit hinzu.

6. Wenn Sie sich das erzeugte PDF ansehen, erscheint die Zitation noch nicht richtig. Sie müssen `pdflatex` noch zweimal aufrufen. Erstellen Sie ein drittes Target namens `references`, das noch zweimal `pdflatex` aufruft. Achten Sie darauf, dass `doc2.bbl` bereits existieren sollte, damit `references` ausgeführt wird. Fügen Sie `references` auch als Abhängigkeit zu `make all` hinzu.

7. Löschen Sie alle von pdflatex und bibtex erzeugten Dateien. Wenn Sie jetzt `:~$ make` oder `:~$ make all` ausführen, sollten Sie das Dokument vollständig kompilieren können.

8. Zuletzt sollen Sie Ihr Verzeichnis aufräumen. Erstellen Sie ein Target, das mit `:~$ make clean` aufgerufen wird und alle von pdflatex und bibtex erzeugten Dateien entfernt (Entfernen Sie `.log`, `.aux`, `.toc`, `.bbl`, `.blg` und natürlich `.pdf`-Dateien).

Zusatzaufgabe : die oben erstellte Makefile hat noch einen Haken: Die Datei `doc2.aux` wird jedesmal verändert, wenn `pdflatex` läuft. Das bedeutet, dass das Dokument jedes Mal neu kompiliert wird, wenn Sie `make` aufrufen - selbst wenn sich am Quellcode nichts geändert hat. Um das zu umgehen, können Sie das speziell für LaTeX

entwickelte Paket `latexmk` benutzen. Informieren Sie sich über `latexmk` auf `https://www.ctan.org/pkg/latexmk/` und versuchen Sie, die Makefile so zu ändern, dass sie statt `pdflatex` und `bibtex latexmk` aufruft.

Übung 8: Tree Tagger

1. In dieser Aufgabe wollen wir einen Text mit dem TreeTagger lemmatisieren und die Nomen-Lemmata extrahieren.
   a) Wechseln Sie auf den Rechner ella, gehen Sie in Ihr `Vorkurs-Verzeichnis` und kopieren Sie sich die Datei `/home/public/vorkurs_ws17/Darth_Vader.txt`. Es handelt sich dabei um den ersten Absatz des Wikipedia-Artikels über Darth Vader.
   b) Führen Sie mit `source` das setup-Skript für den TreeTagger aus.
   c) Benutzen Sie eine Pipe, um den Text durch den Tree-Tagger zu schicken. Da es ein englischer Text ist, sollten Sie `tree-tagger-english` verwenden.
   d) Filtern Sie aus der Ausgabe des TreeTaggers nun einige Wörter heraus, so dass nur noch Nomen (POS-tag: NN oder NNS) übrigbleiben.
   e) Wie Sie nun sehen können, wird für einige Nomen kein Lemma gefunden. TreeTagger gibt an der Stelle ein `<unknown>` aus. Werfen Sie nun aus der Ausgabe alle Zeilen heraus, in denen ein unknown vorkommt.
   f) Entfernen Sie bitte die ersten beiden Spalten, so dass nur noch das Lemma auf jeder Zeile steht.

2. In der nächsten Aufgabe sollen Sie das tree-tagger-Programm direkt benutzen. Es heißt `tree-tagger`. Die wichtigste Option, die das Programm bekommt, ist das statistische Modell. Es steckt in einer sog. par-Datei (für englisch: `/resources/processors/tagger/tree-tagger/lib/english.par`).
   a) Rufen Sie tree-tagger zunächst nur mit der par-Datei auf und füttern Sie ihn über eine Pipe mit einzelnen Wörtern Ihrer Wahl. Das tree-tagger-Programm kann nur einzelne Wörter verarbeiten.
   b) Spielen Sie mit den Optionen herum, die tree-tagger anbietet.
   c) Da Sie ja mittlerweile wissen, wie man Texte so umformatiert, dass jedes Wort auf einer einzelnen Zeile steht – tun Sie das mit obigem Text und füttern Sie ihn direkt in das tree-tagger-Programm (*nicht* in das shell-Skript `tree-tagger-english` sondern das Programm `tree-tagger`).
   d) Erstellen Sie eine Lexikon-Datei, in der Sie die unbekannten Wörter manuell taggen (es gibt eine Beispiel-Lexikon-Datei: `lib/english-lexicon.txt`). Einige unbekannte Wörter: Episode, Anakin, prequel, …
   e) Taggen Sie nun noch einmal `Darth_Vader.txt`, aber übergeben Sie dem tree-tagger Ihr manuell erstelltes Lexikon. Sie sollten jetzt keine `<unkown>` mehr sehen.

Übung 9: Malt Parser

1. Finden und aktivieren Sie den MaltParser in unseren Ressourcen! Welches ist die neueste Version, die bei uns installiert ist?

2. Starten Sie den MaltParser ohne Argumente.

3. Trainieren Sie ein Parsing-Modell mit der Trainingsdatei `examples/data/talbanken05_train.conll`. Wie heißt Ihre Modell-Datei?

4. Testen Sie Ihr Parsing-Modell mit der Testdatei `examples/data/talbanken05_test.conll` und speichern Sie das Ergebnis in einer Datei.

Übung 10: Stanford Parser

1. Finden und aktivieren Sie den Stanford Parser in unseren Ressourcen! Welches ist die neueste Version, die bei uns installiert ist?

2. Starten Sie den Stanford Parser ohne Argumente.

3. Wenn Sie mit einem Browser die Datei `index.html` im Verzeichnis des Parsers öffnen, können Sie die Dokumentation aufrufen. Machen Sie sich mit Hilfe der Dokumentation der `LexicalizedParser`-Klasse mit der Bedienung des Parsers vertraut.

4. Parsen Sie nun die Datei `Darth_Vater.txt` mit dem Stanford Parser. Verwenden Sie dazu das vortrainierte Modell `englishPCFG.ser.gz` im `edu/stanford/nlp/models/lexparser/` Verzeichnis des Stanford Parsers. Probieren Sie verschiedene Ausgabeformate aus. Erstellen Sie eine Datei `Darth_Vader.trees` im Ausgabeformat `penn`.

5. Konvertieren Sie `Darth_Vader.trees` ins `conll`-Format.

Zusatzaufgabe: Parsen Sie die `.conll`-Datei mit dem MaltParser und vergleichen Sie die Ausgabe des MaltParsers mit der Ausgabe des Stanford Parsers.

Übung 11: NLTK

1. Starten Sie eine Python Interpreter-Session (`$ python` oder `$ ipython`) und importieren Sie zunächst nur das `corpus` Modul.

2. Wir wollen genauer wissen, was für Korpora uns das NLTK bietet. Sie sind alle im Modul `corpus` enthalten. Versuchen Sie, herauszufinden, welche Korpora enthalten sind und finden Sie eines, das in getaggter Form vorliegt. Tipp:

   - Die Korpora sind unterschiedlich formatiert und annotiert. Welche Formate und Annotationen ein Korpus bietet, erkennt man schnell daran, welche Methoden sein Reader enthält (z.B. in der API oder über die Onlinehilfe und Autovervollständigung in iPython). Eine Übersicht gibt es im Appendix des NLTK-Buches.

   a) Lassen Sie sich die Anzahl der im Corpus enthaltenen Wörter ausgeben.

   b) Lassen Sie sich die Anzahl der im Corpus enthaltenen Sätze ausgeben.

3. Wir wollen jetzt sehen, welche Möglichkeiten das NLTK für den Umgang mit CFGs bietet und dazu die Module `tree` und `grammar` näher betrachten. Importieren Sie einfach das gesamte NLTK in Ihre Python Session. Nehmen wir an, wir wollen aus einem geparsten Korpus eine Grammatik induzieren.

   a) Finden Sie heraus, wie ein Parsebaum als String aussehen muss, damit das `nltk.tree` Modul ihn in ein Objekt der `Tree`-Klasse umwandeln zu können.

   b) Erstellen Sie einen Parsebaum in diesem Format für einen Beispielsatz. Nehmen Sie dazu einen der Sätze des Korpus oder wählen Sie diesen: `"Einstein war ein Physiker."`

   c) Studieren Sie die API-Dokumentation oder die Onlinehilfe, um herauszufinden, mit welcher Funktion von `nltk.tree.Tree` Sie diese Stringrepräsentation in ein `Tree`-Objekt überführen können. Erzeugen Sie so ein Objekt für den Beispielsatz und weisen Sie ihm einen Namen zu, z.B. `my_tree`.

   d) Lassen Sie sich `my_tree` graphisch darstellen. Dazu hat jede `Tree` Instanz eine eingebaute Methode. Sie können ihre Analyse so leicht überprüfen.

   e) Praktisch an der Klasse `Tree` ist, dass sie gleich die benutzten Regeln im Baum ermittelt. Lassen Sie sich die Produktionsregeln von Ihrem Baum ausgeben. Auch das geht mit einer Instanz-Methode.

   f) An dieser Repräsentation ist wiederum praktisch, dass man sie direkt weiter verwenden kann, um eine Grammatik zu erstellen. Finden Sie über die API des `grammar` Moduls dazu die richtige Klasse. Erzeugen Sie die aus Ihrem Satz induzierte (sehr kleine) kontextfreie Grammatik.

   g) Falls Sie den obigen Beispielsatz verwendet haben, sollte die Grammatik jetzt die Tokensequenz `` `ein Physiker' `` abdecken. Wie finden Sie heraus, ob sie das tut?

4. Um nun eine etwas umfangreichere Grammatik zu erzeugen, benutzen wir die Penn Treebank. Das `corpus treebank` liegt u.A. als Liste von Parse-Bäumen vor.

   a) Schreiben Sie eine Funktion `getCFG()`, die eine Liste aller in der `treebank` benutzten Produktionsregeln erzeugt (behandeln Sie hierbei Mehrfachvorkommen sinnvoll) und anschließend daraus eine CFG erstellt und diese zurückliefert.

   b) Sehen Sie sich das `parse` Modul an. Es bietet Implementationen verschiedener CFG-und PCFG-Parser. Wählen Sie einen, der mit einer CFG arbeitet. Schreiben Sie eine Funktion `drawParses()`, die eine CFG und einen Satz als Argumente nimmt. Sie soll dann einen entsprechenden Parser erzeugen, den Satz parsen und die drei besten Bäume grafisch ausgeben. Probieren Sie zunächste eine kurze Phrase wie `"British ships"`, da je nach Parser bei längeren Sätzen die Gefahr besteht, in eine Endlosschleife zu geraten. Hinweis:

      - Manche Parser benötigen die Angabe einer Strategie. Dazu am besten die Variable `nltk.parse.chart.BU_STRATEGY` übergeben. Es ist eine Sammlung von Regeln, nach denen die verschiedenen Expansionen durchprobiert werden.

Übung 12: spaCy

1. Starten Sie eine Python Interpreter-Session (`$ python` oder `$ ipython`).
2. Importieren Sie die spaCy-Bibliothek.
3. Laden Sie das spaCy-Modell für Englisch.
4. In dieser Aufgabe wollen wir die Performanz von spaCys Tagger und dem Standard-Tagger von NLTK vergleichen. Hierzu benutzen wir den in NLTK erhaltenen Brown-Korpus.
   a) Benutzen Sie NLTK, um den Brown-Korpus zu laden.
   b) Extrahieren Sie alle Wörter aus dem Brown-Korpus.
   c) Wenden Sie NLTKs bevorzugten Tagger auf die Liste der Wörter an. Auf diesen Tagger können Sie mit `nltk.pos_tag` zugreifen.
   d) Wenden Sie spaCys Tagger auf die Wörter an.
   e) Extrahieren für beide Ausgaben eine Liste der zugewiesenen Tags (in spaCy benutzen Sie hierfür das Attribut `tag_`).
   f) Extrahieren Sie die Wörter mit den annotierten Tags im Brown-Korpus, indem Sie die `tagged_words`-Methode des Korpus benutzen.
   g) Erstellen Sie eine Liste der annotierten Tags.
   h) Berechnen Sie mit `nltk.accuracy` die Accuracy der Ausgaben, indem Sie diese mit den annotierten Tags vergleichen.

Übung 13: Profiler

1. Kopieren Sie sich den Ordner `/home/public/vorkurs_ws18/tinysearch` in Ihr `Vorkurs`-Verzeichnis. Der Ordner enthält die Implementierung unserer einfachen „Suchmaschine". Schauen Sie sich den Code nochmal an und führen Sie `tiny_search_engine.py` aus.

2. Messen Sie die Laufzeit mit `cProfile` für verschiedene Werte von `vocab-size` und `num-docs`.

3. Lassen Sie nun `tiny_search_engine.py` mit dem `cProfiler` und den Optionen `-v 1000 -d 1000` laufen, schreiben Sie die Profiler-Statistiken in eine Datei und erstellen Sie den Callgraph.

4. Verwenden Sie jetzt den Line Profiler, um sich die Funktion `cosine(a,b)` im Modul `tinysearch/cosinus.py` genauer anzuschauen. Welche Zeilen benötigen die meiste Laufzeit?

Übung 14: numpy

1. Löschen Sie den Ordner `tinysearch` in Ihrem Vorkurs-Verzeichnis und kopieren Sie sich die neue Version von `/home/public/vorkurs_ws18/tinysearch`.

2. Jetzt sollen Sie die Kosinusähnlichkeit mit NumPy und SciPy implementieren:

   a) In `tinysearch/cosinus.py` finden Sie eine unvollständige Funktion `cosine_numpy(a,b)`. Implementieren Sie die Kosinusähnlichkeit jetzt mit Hilfe von NumPy-Funktionen und Datenstrukturen, und geben Sie diese zurück. Sie können davon ausgehen, dass `a` und `b` bereits `ndarrays` sind. Benutzen Sie hierbei nicht die SciPy-Library.

   b) Ändern Sie jetzt in `tiny_search_engine.py` die `main`-Funktion so, dass `run_numpy()` statt `run()` verwendet wird. Evaluieren Sie mit dem Line Profiler Ihre Funktion und vergleichen Sie sie mit der Pure Python-Variante.

   c) Finden Sie heraus, ob es in SciPy bereits eine Implementierung des Kosinusabstands gibt. Falls ja, vervollständigen Sie die Funktion `cosine_scipy()` in `cosinus.py`. Ändern Sie in `tiny_search_engine.py` dann die Funktion `run_numpy()` so ab, dass Sie `cosine_scipy()` aufruft, und messen Sie nochmals die Laufzeit.

Übung 15: scikit-learn

1. Genau wie NLTK stellt auch `scikit-learn` einige Datensets bereit. In dieser Übung sollen Sie mit dem 20newsgroups-Datenset arbeiten. Es enthält 20,000 Newsgroup-Dokumente aus 20 verschiedenen Kategorien enthält. Jedes Dokument ist genau einer Kategorie zugeordnet, die im Datenset annotiert ist. Das Datenset ist in Trainings- und Testdaten unterteilt, wobei alle Kategorien in Train und Test ungefähr gleich proportioniert vorkommen.

   a) Laden Sie nun die Trainingspartition von 20newsgroups. Sehen Sie sich dazu die Dokumentation der Funktion `sklearn.datasets.fetch_20newsgroups` an. Laden Sie die Trainingsdaten in eine Variable `newsgroups_train`.

   b) Die Dokumente sind nun in `newsgroups_train.data` als Liste gespeichert. Finden Sie heraus, wie viele Dokumente im Trainingsset sind.

2. Jetzt sollen Sie sich ein Trainingsset bauen, das nur Dokumente aus zwei Kagorien enthält und die Dokumente in Feature-Vektoren transformieren.

   a) Die Kategorien der Dokumente sind als Integer-IDs in `newsgroups_train.target` gespeichert. Überzeugen Sie sich, dass es genau so viele Kategorien wie Dokumente sind.

   b) `newsgroups_train.target_names` speichert die Namen der Kategorien. Lassen Sie sich die Kategorien anzeigen.

   c) Wählen Sie sich zwei Kategorien aus, die Ihnen nicht zu ähnlich erscheinen. Verwenden Sie dann nochmal `fetch_20newsgroups`, um nur die Trainingsdaten zu laden, die zu diesen beiden Kategorien gehören. Speichern sie diese in einer Variable `newsgroups_train_2cat`, und finden Sie heraus, wie viele Dokumente geladen wurden.

   d) Erstellen Sie jetzt für Ihr Trainingsset eine Dokument-Term-Matrix aus den Trainingsdaten (1 Reihe pro Dokument). In der Vorlesung haben wir die Klasse `sklearn.feature_extraction.text.CountVectorizer` verwendet. Man kann allerdings auch gewichtete Counts extrahieren. Verwenden Sie nun die Klasse `TfidfVectorizer`.

3. Jetzt können Sie einen Klassifizierer trainieren und auf Testdaten evaluieren.

   a) Verwenden Sie das „Flowchart" aus den Slides, um nach einem passenden Algorithmus zu suchen.

   b) Verwenden Sie den Algorithmus, um einen Klassifizierer zu trainieren, indem Sie die Dokument-Term-Matrix als Beobachtungen und die Kategorien (`newsgroups_train_cat2.target`) als Labels behandeln.

   c) Verwenden Sie jetzt wieder `fetch_20newsgroups()`, um die Testdaten aus denselben Kategorien zu laden. Erstellen Sie wieder eine Dokument-Term-Matrix `X_test`. Verwenden Sie nun das trainierte Modell, um die Testdaten zu klassifizieren. *Hinweis*: Verwenden Sie zum Vektorisieren dasselbe Objekt, mit dem Sie auch die Trainingsdaten vektorisiert haben. Rufen sie *nicht* noch einmal die `fit()`-Funktion auf, da sich sonst das Vokabular ändern würde.

   d) Evaluieren Sie die Accuracy Ihres Klassifizierers, indem Sie die Gold-Labels (`newsgroups_test_2cats.target`) mit den vom Klassifizierer ausgegebenen vergleichen. Schauen Sie in `sklearn.metrics` nach der Evaluierungsmetrik.

4. Wenn noch Zeit ist, können Sie folgendes ausprobieren:

   - Trainieren Sie ein Modell für zwei Kategorien, die sich sehr ähnlich sind
   - Trainieren Sie ein Modell mit mehr als zwei Kategorien!
   - Probieren Sie einen anderen Klassifizierer aus.

Übung 16: Weka

1. Kopieren Sie die Datei `/home/public/vorkurs_ws18/playoutside.arff` in Ihr Vorkurs-Verzeichnis. Der Datensatz enthält verschiedene Wetterbedingungen zusammen mit der Angabe, ob man draußen spielen kann oder nicht.

2. Schauen Sie sich die Datei an. Die Datei enthält die Wetterdaten Vorhersage (outlook), aktuelle Temperatur (temperature, in Fahrenheit), Luftfeuchtigkeit (humidity) und Windverhältnisse (windy, TRUE oder FALSE, also ein boolescher Wert). Als letztes enthält die Datei außerdem das Datum "play", das angibt ob man spielen kann oder nicht. Es handelt sich also um *a*nnotierte Daten.

   Möchte man einen Classifier produktiv einsetzen (um echte Daten zu klassifizieren) braucht man natürlich auch unannotierte Daten, die man klassifizieren möchte. Wir werden später solche Daten erstellen. Wichtig ist, dass die annotierten und unannotierten Datensätze die gleichen Daten enthalten – abgesehen vom letzten Datum, das die Klasse beschreibt.

3. Wir werden nun zunächst einen geeigneten Algorithmus suchen. Dazu verwenden wir ausschließlich die annotierten Daten, da wir damit auch gleich evaluieren können, wie gut der Algorithmus funktioniert hat. Wichtig ist generell, dass man nicht die gleichen Daten zum trainieren und testen verwendet (Weka sorgt dafür, dass uns das nicht passiert).

   Aktivieren und starten Sie Weka 3.7.7 (zu finden im Verzeichnis `/resources/stat_ml/weka-3.7.7`). Als erstes öffnet sich der GUI Chooser. Wählen Sie den Explorer, öffnen Sie die Datei. Weka zeigt Ihnen zunächst eine genauere Analyse der Daten an (welche Werte kommen wie oft vor etc.). Schauen Sie sich die Daten genau an.

4. Wählen Sie dann den Reiter "Classify" und probieren Sie einige Classifier aus. Einige können nicht mit numerischen Werten umgehen, andere nicht mit nominalen. Entsprechende Fehlermeldungen weisen Sie darauf hin[1]. Lassen Sie die Test options auf 10 fold cross-validation eingestellt. Merken Sie sich den Classifier, der die besten Ergebnisse liefert.

5. Erstellen Sie nun eine Datei in Ihrem Vorkurs-Verzeichnis, die z.B. `Daten.arff` heißt. Darin sollen neue annotierte Daten stehen, die Sie sich selbst ausdenken dürfen. Es reicht, wenn ein bis zwei Datensätze drinstehen. Wichtig: Sie müssen den Header (alles vor der `@data`-Markierung) in die Datei schreiben (oder kopieren).

6. Wechseln Sie wieder in das Weka-Fenster und den Reiter "Classify". Wählen Sie nun unter test options "Supplied test set", klicken Sie auf set und sorgen Sie dafür, dass die eben erstelle Arff-Datei zum testen verwendet wird. Klassifizieren Sie mit dem gleichen Classifier wie davor.

[1] Kaum ein Classifier kann mit String-Werten umgehen.

7. Unter “More Options” gibt es weitere Optionen zur Klassifizierung. Aktivieren Sie dort “Output predictions”. Damit werden die tatsächlichen Vorhersagen für einzelne Datensätze ausgegeben. Klassifizieren Sie erneut und schauen Sie sich die einzelnen predictions an.

8. Weka lässt sich auch in der Kommandozeile verwenden. Wechseln Sie wieder in Ihr Terminalfenster. Verwenden Sie den kompletten (qualifizierten) Namen des Classifiers als Java-main-class (z.B. `:~$ java weka.classifiers.trees.J48`). Wenn Sie keine Optionen angeben, bekommen Sie eine Hilfeseite angezeigt, die die möglichen Optionen listet. Versuchen Sie die die gleiche Ausgabe wie auch in der GUI zu bekommen.

Übung 17: Cluster

1. Verbinden Sie sich mittels `ssh cluster` mit dem Cluster
2. Schauen Sie sich an, welche Partitionen und Nodes es gibt. Sind Nodes momentan nicht verfügbar?
3. Zählen Sie, wie viele Jobs momentan auf der Partition `gpulong` laufen.
4. Allozieren Sie mit `salloc` Ressourcen für einen Job auf der main-Partition. Geben Sie die folgenden Optionen an:
   - Der Task soll maximal zehn Minuten laufen
   - Wir wollen nur eine CPU pro Task verwenden
   - Wir wollen mindestens auf zwei Nodes laufen
   - Insgesamt wollen wir vier Prozessoren verwenden
5. Rufen Sie `sacct` auf und schauen Sie sich ihren Job an. Welche ID hat er?
6. Führen Sie mit `srun` das Kommando `hostname` auf den allozierten Nodes aus. Welche Ausgabe erhalten Sie? Warum?
7. Führen Sie das Kommando ohne `srun` aus. Was ist anders?
8. Beenden Sie ihren Job durch `scancel`
9. Verwandeln Sie die oben ausgeführten Aufrufe in ein Batch-Script und führen Sie es aus. Übertragen Sie alle Optionen, die Sie zuvor bei `salloc` verwendet haben. Fügen Sie außerdem Anweisungen hinzu, damit Sie über den Fortschritt des Jobs per Mail informiert werden. Welche Nachrichten bekommen Sie?
10. Wir wollen nun einige Dateien auf dem Cluster verarbeiten. In der Datei `/home/public/vorkurs_ws18/nltk-tagger` finden sie einen simplen Tagger, den wir auf dem Cluster ausführen wollen. Kopieren Sie ihn auf das Cluster.
11. Neben dem Tagger brauchen wir auch Daten. Diese haben die Form `kant.txt-split0x` und finden sich im selben Verzeichnis. Die Daten sind Teilstücke eines größere Korpus und wurden aufgeteilt, damit wir sie parallel verarbeiten können. Kopieren Sie sie auch auf das Cluster.
12. Erstellen Sie eine Python 2 virtualenv auf dem Cluster und aktivieren Sie sie.
13. Installieren Sie die benötigten NLTK-Werkzeuge, indem Sie in Python folgende Befehle ausführen:

```
import nltk
nltk.download('averaged_perceptron_tagger')
nltk.download('punkt')
```

14. Verwenden Sie ein Job Array, um Jobs zu erstellen, die den Tagger jeweils einmal auf jede Datei anwenden. Der Aufruf für das Programm lautet dabei:

```
python nltk-tagger.py <input-file>
```

15. Der Tagger hat für jede Eingabedatei eine Ausgabedatei erstellt. Kopieren Sie sie alle zurück in ihr (Pool-)Home-Verzeichnis und fügen Sie sie zusammen.