Finding Structure in Pitching Patterns

Matthew Lawson

June 11, 2015

Abstract

Humans are remarkable bad at generating random sequences. One such non-random sequence is the sequence of pitch types generated by a baseball pitcher. Modern electronic data collection and records-keeping has made the full time-series of pitch types thrown by major-league pitchers since 2008 available for analysis. In this report I present the results of a computational mechanics analysis of pitching time-series.

1 Data Acquisition

The data I used was originally collected by the MLB, with a network of cameras and computer algorithms, classifying pitch types. The data is released on the MLB web-site in XML files. I used the R program Pitchrx to collect the data and place it in a MySQL database. I also used some data from a database (generated from the same base dataset) available at baseballheatmaps.com. I also spent a substantial amount of time fiddling with a set of perl scripts and a python module to collect and query this data. All of the available code seems to be of a very low quality. Pitchrx is the only actively maintained of the modules I worked with, and it unfortunately generates a poorly thought out database. It did. however, prove to be the best option.

2 Data Processing

I wished to do my analysis in python, so interfacing with the database from a python context was the next step. I first tried using the ORM libraries SQLAlchemy and peewee, however the design of the Pitchrx database was such that it made working with an ORM very difficult. Eventually I reverted to the MySQLdb python module, and writing raw SQL queries as long strings. This proved much more effective.

Then I moved to considering the question of how best to use the data such that it would actually display patterns. The easiest thing, of course, is treating each pitch as an event in a single time-series, without regards to the fact that there exists different at-bats, innings, and games. One would expect a pitcher to behave different when facing right-handed vs left-handed pitchers, however. I thus elected to consider only right handed

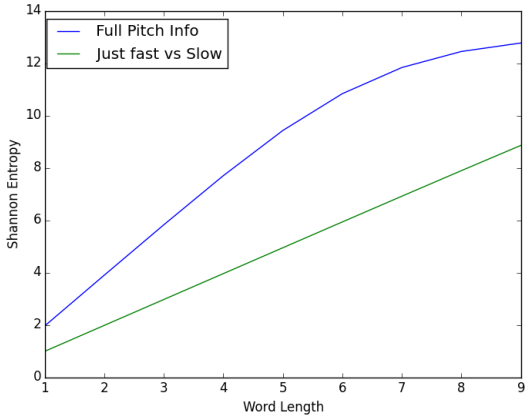

Figure 1: The block entropies for binary and full-alphabet pitch sequences

hitters. It was straightforward to write SQL queries to generate ordered time series of pitches for a specific handedness (see the python function get_ts_for_handed() in Appendix A). I did some analysis of this time-series, which results will be discussed below.

One could also condition on individual match-ups (that is, one pitcher, one hitter), whoever the data set is generally sufficiently limited as to make this particular analysis infeasible.

One could also divide the data up into a collection of short time-series, based on game, inning, or at-bat. I did so for at-bat (generating time-series with lengths 1-14). The CMPy code however, was not equipped to deal with this type of data. I'll discuss modifications required later, although you can see the full modified code in Appendix B.

Given that any given pitcher probably has a repertoire of 5 pitches or so, it makes sense to use a 5-letter alphabet to represent the process. However, since the compute time for Bayesian Inference scales combinatorially with the alphabet size, it might also be useful to consider a binary partitioning of the data. Generally, pitches can be classified as fastballs or off-speed pitches, so I also wrote python functions to return a time-series of pitches divided into a two character (1s and 0s) alphabet depending on pitch speed (see the python function separate_by_speed()).

3 Entropy Rate

Using the DIT python package, I calculated entropy rates of both the binary and full 5-pitch time-series (not divided by at-bat) until such time as my computer ran out of memory and crashed. The results can be seen in figure 1.

The roll-off in the 5-character alphabet is probably due to the effects of finite alphabet size, however you can easily see that the entropy rate

of the binary partition is below 1 (that is, below $\log_2(2)$, and it looks like the asymptotic entropy rate of the 5-character alphabet will be below $\log_2(5) = 2.3$. This indicates that there is indeed some sort of structure or predictability to the time-series.

4 Tree-merging Algorithms

I made several attempts to use the CMPy Topological Merger codes to generate an epsilon machine from the raw time-series, the binary time-series, and the list of short at-bat time-series. This last attempt required some light modifications of the existing code, to allow the introduction of a word distribution, rather than a long time-series, and development of my own code to generate a distribution from short time-series. The code to do this is included in the Appendices, but is straightforward.

Unfortunately, these efforts were in vain. The pitch time-series to do not have forbidden words, (they have full support) and thus are topologically indistinguishable from a biased coin. The CMPy codes for probabilistic tree-mergers are not currently in working order, and I did not have time to investigate them.

5 Bayesian Inference of Epsilon Machines

The natural next step is to use the Bayesian Inference codes. When run on the raw time-series of binary data, the result is a biased coin, which suggest that whatever structure exists in the pitch series is destroyed by concatenation (which is not surprising). I thus undertook to modify the Bayesian Inference code to accommodate lists of short samples of data. What I attempted was to modify where the code traces the path the data takes through each proposed machine structure. The addition of an additional loop over short data snippets allowed the code to work, although I did not attempt to prove the correctness of this modification (although I see no reason for it to be incorrect).

The result of inference over machines of 5 or less states on a binary alphabet of short at-bats is shown in figure 2 with probability 30%.

Inference over the full 5-pitch alphabet has yet to succeed due to the extreme memory requirements of such a computation.

6 Future Work

There is really quite a bit more work to do on this project. Future work includes Bayesian inference on the full 5-character alphabet, comparisons of structures produced by different pitchers, doing probabilistic tree-merger, doing the above on time-series divided by inning or game, and full analysis of the machine produced by the Bayesian inference code.

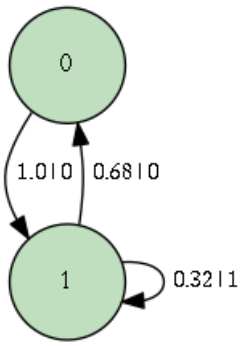

Figure 2: The machine inferred by the Baysian method for the binary time-series

7 Appendix A: Code

```
#!/usr/bin/python2

import collections
import networkx
import numpy as np
import scipy as sp
import MySQLdb
import resource
import sys
import cmpy
import cmpy.inference
import dit
import dit.inference
import cmpy.inference.bayesianem as bayesem


# Set higher recursion limits for our recursive function
sys.setrecursionlimit(10**6)
resource.setrlimit(resource.RLIMIT_STACK, (2**29, -1))


conn = MySQLdb.connect(user='matthew', passwd="pitchfx", db="pitchfx")
curs = conn.cursor()
forbidden_pitches = [None, "IN", "PO", "FO", "XX", "UN"]
pitch_types = {"FA": 0,
               "FF": 1,
               "FT": 2,
               "FC": 3,
               "CU": 4,
               "SI": 5,
               "SF": 6,
               "SL": 7,
```

```
                "CH": 8,
                "CB": 9,
                "FS": 10,
                "KC": 11,
                "KN": 12,
                "EP": 13,
                "UN": -1,
                "XX": -1,
                "PO": -1,
                "FO": -1,
                "IN": -1
                }


def get_ts_for_pitcher(pitcher_name):
    query = (" SELECT pitch_type, pitch.inning, pitch.num, tfs_zulu "
             "FROM pitch "
             "INNER JOIN atbat "
             "ON pitch.gameday_link = atbat.gameday_link "
             "AND pitch.num = atbat.num "
             "WHERE pitcher_name = '{}' "
             "AND tfs_zulu != '' "
             "ORDER BY pitch.tfs_zulu").format(pitcher_name)
    curs.execute(query)
    # now clean out the Nones
    pitches = curs.fetchall()
    # remove Intentional Walks and Nones
    pitches = filter(lambda x: x[0] not in forbidden_pitches, pitches)
    return list(pitches)


def get_ts_for_matchup(pitcher_name, batter_name):
    query = (" SELECT pitch_type, pitch.inning, pitch.num, tfs_zulu "
             "FROM pitch "
             "INNER JOIN atbat "
             "ON pitch.gameday_link = atbat.gameday_link "
             "AND pitch.num = atbat.num "
             "WHERE pitcher_name = '{}' "
             "AND batter_name = '{}' "
             "AND tfs_zulu != '' "
             "ORDER BY pitch.tfs_zulu").format(pitcher_name, batter_name)
    curs.execute(query)
    # now clean out the Nones
    pitches = curs.fetchall()
    # remove Intentional Walks and Nones
    pitches = filter(lambda x: x[0] not in forbidden_pitches, pitches)
    return list(pitches)


def get_ts_for_handed(pitcher_name, handedness):
```

```
    query = (" SELECT pitch_type, pitch.inning, pitch.num, tfs_zulu "
             "FROM pitch "
             "INNER JOIN atbat "
             "ON pitch.gameday_link = atbat.gameday_link "
             "AND pitch.num = atbat.num "
             "INNER JOIN player "
             "ON atbat.batter = player.eliasid "
             "WHERE pitcher_name = '{}' "
             "AND player.throws = '{}' "
             "AND tfs_zulu != '' "
             "ORDER BY pitch.tfs_zulu").format(pitcher_name, handedness)
    curs.execute(query)
    # now clean out the Nones
    pitches = curs.fetchall()
    # remove Intentional Walks and Nones
    pitches = filter(lambda x: x[0] not in forbidden_pitches, pitches)
    return list(pitches)


def seperate_by_ab(pitcher_timeseries, binary=False):
    """takes a pitch timeseries where each pitch is of the form
    (pitch_type, pitch.inning, pitch.num, tfs_zulu)
    and returns a list of lists, each list is the pitch_types for a particular
    at_bat.
    """
    atbats = []
    atbat = []
    for num, pitch in enumerate(pitcher_timeseries):
        pitch = list(pitch)
        # fix misclassifified pitches, a hack for matt cain
        if pitch[0] == 'FA':
            pitch[0] = 'FF'

        if binary:
            if pitch[0][0] == "F":
                pitch[0] = '1'
            else:
                pitch[0] = '0'

        my_map = {'CH': '0', 'CU': '1', 'FF': '2', 'FT': '3', 'SL': '4'}
        pitch[0] = my_map[pitch[0]]

        if num == 0:
            atbat.append(pitch[0])
            continue
        if pitcher_timeseries[num - 1][2] == pitch[2]:
            atbat.append(pitch[0])
        else:
            atbats.append(atbat)
            atbat = [pitch[0]]
```

```
    return atbats


def seperate_by_speed(pitches):
    """takes a nested list of any depth and replaces leading length 2 strings
    desegnating fastball pitches with 1 and those desegnating off speed pitches
    with 0.
    """
    pitches = list(pitches)
    if pitches == []:
        return []
    if type(pitches[0]) == str and len(pitches[0]) == 2:
        if pitches[0][0] == "F":
            pitches[0] = '1'
        else:
            pitches[0] = '0'
        return pitches

    # recursive step:
    first = seperate_by_speed(pitches[0])
    rest = seperate_by_speed(pitches[1:])
    return [first] + rest


def bayes_infer(bb_ts, symbols, max_machine_size):
    """Do baysian inference. symbols is the number of symbols in the alphabet.
    max_machine_size is the maximum number of allowed states."""
    # get set of 1- to 4-state topological epsilon machines
    # * first argument is the alphabet size; 2 is a binary alphabet
    # * the second is a list of number of states,
    # we want all 1-state, 2-state, 3-state and 4-state machines
    #   so we provide [1,2,3,4]
    modelset1 = bayesem.LibraryGenerator(
        symbols, range(1, max_machine_size + 1), em_min='none')

    # declare prior over models
    prior = bayesem.ModelComparisonEM(modelset1, beta=4., verbose=True)

    # get InferEM instance from prior
    # * this actually returns a list
    # (in case there are more than one most probable topologies)
    # * we consider the first element of this list
    # * each element of the list is a tuple:
    # (probability of machine, machine InferEM instance)

    pr, prior_inferem = prior.get_MAP_InferEM()[0]

    # get set of 1- to 4-state topological epsilon machines -- same as the
    # prior above
    modelset2 = bayesem.LibraryGenerator(
```

```
        symbols, range(1, max_machine_size + 1), em_min='none')

    # declare posterior over models
    posterior = bayesem.ModelComparisonEM(
        modelset2, bb_ts, beta=4., verbose=True)

    pr, post_inferem = posterior.get_MAP_InferEM()[0]

    # sample a machine-- returns startnode and machines
    sn, post_sample = post_inferem.generate_sample()
    print('start node: ', sn)

    # draw
    return (pr, post_inferem)


def get_info_meas(depth):
    mcrh = get_ts_for_handed("Matt Cain", "R")
    mcb = seperate_by_speed(mcrh)
    # just the pitches
    mcrh = [i[0] for i in mcrh]
    mcb = [str(i[0]) for i in mcb]

    full_ent = []
    for i in range(1, depth):
        print("starting length: ", i)
        dist = dit.inference.distribution_from_data(mcrh, i)
        full_ent.append(dit.shannon.entropy(dist))
        print("Done!")

    speed_ent = []
    for i in range(1, depth):
        print("starting length: ", i)
        dist = dit.inference.distribution_from_data(mcb, i)
        speed_ent.append(dit.shannon.entropy(dist))
        print("Done!")

    return (full_ent, speed_ent)


def tree_from_dist(data, root=tuple()):

    ids = collections.defaultdict(lambda: len(ids))
    tree = networkx.DiGraph()
    tree.add_node(root, label=ids[root])

    for word in data:
        parent = root

        for j in range(len(word)):
```

```
            child = tuple(word[:j + 1])
            tree.add_node(child, label=ids[child])
            tree.add_edge(parent, child, output=word[j])
            parent = child

    return tree

def atbats_to_dist(atbats, depth):
    """Tkaes the result of a seperate_by_ab() call, and generates a word
    frequency distribution over it to a specified depth. It ignores atbats
    shorter than depth. It returns a dictionary with words mapped to
    probabilitites
    """

    dist = {}
    for ab in atbats:
        #first, strip off everything but the pitch designation
        ab = [i[0] for i in ab]

        ab_len = len(ab)
        # only consider atbats long enough to have words to contribute
        if ab_len < depth:
            continue

        #First, count up the instances of each word
        for i in range(ab_len - depth + 1):
            # we have to use a tuple here, so we can use it as a dict index
            word = tuple(ab[i:i + depth])
            try:
                dist[word] += 1
            except KeyError:
                dist[word] = 1

    tot_words = sum(dist.itervalues())

    for i in dist.iterkeys():
        dist[i] /= float(tot_words)

    return dist


def main(bayes=False, tree_depth=5, morph_length=3):
    mcrh = get_ts_for_handed("Matt Cain", "R")
    if bayes:
        mcrh = seperate_by_ab(mcrh, binary=False)
        my_map = {'CH':'0', 'CU': '1', 'FF': '2', 'FT': '3', 'SL': '4'}
        #mcb = seperate_by_speed(mcrh)
        #mcrh = [i[0] for i in mcrh]
        #mcb = [str(i[0]) for i in mcb]
        return bayes_infer(mcrh, 5, 3)
```

```
else:
    mcab = seperate_by_ab(mcrh)
    dist = atbats_to_dist(mcab, tree_depth)
    tree = tree_from_dist(dist)
    return cmpy.inference.TopologicalMerger(None, morph_length=morph_length,
                                            tree_depth=tree_depth, tree=tree)
```

8 Appendix B: Modifications to CMPy

In counts.py, in the _generate_counts() function, starting on line 176

```
        for d in data:
            #check if we have a long stream of data (each d is a string)
            # or if each d is an iterable (in which case our data is a
            # collection of very short time series)

            # because we have nested for-loops, we need to do a sort of
            # thing where we break from the 2nd for loop on a forbidden
            # transition.
            # Check if the forbidden_transition flag is set

            if forbidden_transition:
                forbidden_transition = False
                break

            if hasattr(d, '__iter__'):
            print("made it")
                # We are dealing with a list of short time series data.
                for sub_d in d:
                    # node counts
                    if temp.has_key(currNode):
                        temp[currNode] += 1
                    else:
                        temp[currNode] = 1
                    # node, symbol counts
                    vs = (currNode, sub_d)
                    if temp.has_key(vs):
                        temp[vs] += 1
                    else:
                        temp[vs] = 1
                    # find next node, given previous node, symbol pair
                    if self.eMtrace.has_key(vs):
                        # key exists, this means transition is allowed
                        currNode = self.eMtrace[vs]
```

```
                self.lastNode[startNode] = currNode

                # add to state sequence
                temp_states.append(currNode)
            else:
                # key does NOT exist, forbidden transition
                temp = {}
                temp_states = []
                self.lastNode[startNode] = None
                forbidden_transition = True

    else:
        # node counts
        if temp.has_key(currNode):
            temp[currNode] += 1
        else:
            temp[currNode] = 1
        # node, symbol counts
        vs = (currNode,d)
        if temp.has_key(vs):
            temp[vs] += 1
        else:
            temp[vs] = 1
        # find next node, given previous node, symbol pair
        if self.eMtrace.has_key(vs):
            # key exists, this means transition is allowed
            currNode = self.eMtrace[vs]
            self.lastNode[startNode] = currNode

            # add to state sequence
            temp_states.append(currNode)
        else:
            # key does NOT exist, forbidden transition
            temp = {}
            temp_states = []
            self.lastNode[startNode] = None
            break
```