995 Capstone: Using Machine Learning to Annotate Fingerings for Cello Music

Zach Dunn

1 Introduction

This project explores the application of machine learning to develop an intelligent system capable of recommending optimal fingerings for playing notes on the cello. Given a sequence of musical notes, characterized by their pitch, duration, and offset within each measure, the system will analyze this input to suggest which finger a player should use for each note. Rather than relying on a single algorithm, this project will evaluate multiple machine learning approaches to determine which are most effective for this task. While the ultimate goal is to create a system that generalizes to any piece of music, the primary focus is on beginner-level repertoire, where players are most likely to benefit from fingering assistance due to limited technical experience.

2 Related Work

Some work has been done in this field for piano music as well as violin with varying levels of success.

Gao et al. explored using a Markov decision process (MDP) for modeling the generation of piano fingerings along with prioritized sweeping algorithms for model-based reinforcement learning. The model is comprised of a 4-tuple which holds the state and action spaces as well as the transition probability and reward function [1]. The state of the system was represented by a triplet containing the chronological note number, finger action, and sequence of next notes to play. Additionally, to model the players hand the authors used a matrix that represents valid finger crossings. An algorithm was then used to filter out the invalid fingerings based on what is considered physically possible to do with the hands. To create the reward function, a number of metrics were made to define various difficulty levels for types of hand movements and finger crossings. Lastly, the prioritized sweeping algorithm was used for the reinforcement learning with a few modifications. Instead of creating a matrix that enumerates the entire state space of piano key and fingering positions, Gao et al. created a key-value storage tabular reinforcement learning method that uses a hash function [1]. Looking at the results the authors show that their solution performs better than other methods in many categories tested. They were able to almost completely remove impossible fingerings and achieved the best scores in reducing hand stretching and step spread [1]. These indicate that their method is optimized for motion efficiency.

Iman and Ahn take a slightly different approach to generating piano fingerings using deep reinforcement learning with a Markov decision process as the learning environment. They first define a pianist agent that represents the hand of the player with the goal of finding the optimum fingering for a song. The environment the agent will interact with is defined by four piano pieces where the state is hand position, and the action is choosing a finger [2]. The authors then define the reward function based on a distance matrix that describes the difficulty of positions of finger pairs and 12 fingering rules. Lastly, Iman and Ahn outline the various neural network architectures for the agents they are going to test including a DQN, DDQN, PG, and A2C. To evaluate the methods the authors use two main metrics including difficulty level and match rate. Looking at the results Iman and Ahn found that the DQN method gave the best score for match rate meaning the fingerings were more human-like and it can capture the sequential nature of fingerings [2]. Additionally, the DQN method resulted in the lowest difficulty level for each of the four songs tested.

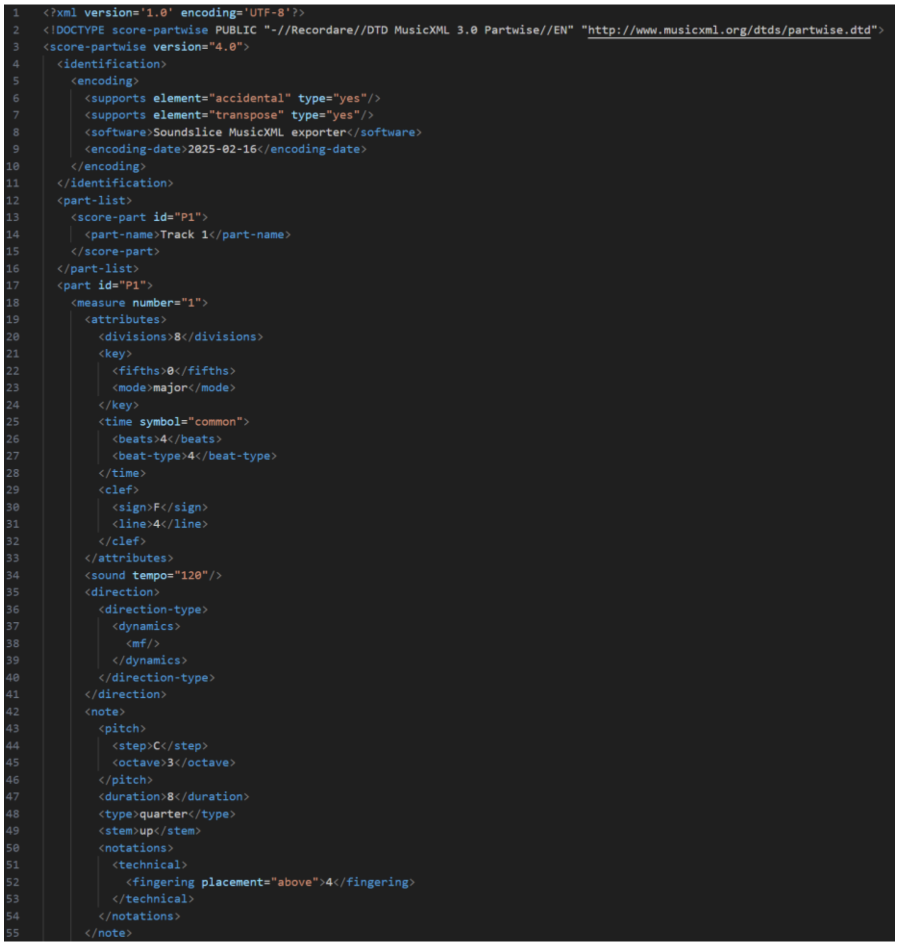
Cheung et al. introduces a new approach to generating fingerings for violin music using limited data and semi-supervised variational autoencoders (VAEs). The architecture of the model the authors created contains four main modules: an embedder, encoder, classifier, and decoder. The input and output of the VAE consists of a MIDI number which describes the note, the onset, and duration of the note. To train the model a dataset of 14 annotated violin songs from a wide range of styles was used. To evaluate the model's performance a number of metrics were used including F1 score, mean reciprocal rank (MRR), and normalized discounted cumulative gain (nDCG). The authors wanted to see how well their model could replicate the style of a performer (correctly predict what fingering a human would do) and to what extent the predicted fingerings were actually used by other performers (preference). Looking at style first with a fully supervised model the authors found that the MRR scores showed that "the true fingerings as performed by the violinist were predominantly given by the model's most probable predictions" [3]. Additionally, the F1 scores showed that the model outperformed previous works though the model unexpectedly performed better for string than finger and position. Cheung et al. also found that the model was able to accurately capture the fingering preferences of human performers since the model was able to achieve high nDCG scores. Finally, when looking at how the model performed in semi-supervised scenarios the authors found that the model was able to learn from unlabeled data. The model was trained on 6 randomly selected unlabeled pieces plus 1 to 7 labeled songs. As a result, the proposed model was able to match the state of the art in terms of fingering generation with half the amount of labeled data [3].

3 Methods

This section is broken down into parts starting with the data collection and pre-processing, followed by individual sections for each machine learning method tested. The following methods were implemented in Python [11] using a collection of packages. The first package, Music21 (https://www.music21.org/music21docs/ version 9.5.0), was used to convert music XML data into a useable data structure, from which the song features could be extracted. Pandas (https://pandas.pydata.org/ version 2.2.3) was used to hold the extracted data in a structure known as a DataFrame. Numpy (https://numpy.org/ version 1.26.4) was used for mathematical operations as well as array manipulation. Keras (https://keras.io/ version 3.8.0) was used for building the machine learning models. Sklearn (https://scikit-learn.org/stable/ 1.6.1) was used for splitting the training data into train and test sets, creating the KNN model, and generating indices for k-fold cross validation. Pretty Midi (https://craffel.github.io/pretty-midi/ version 0.2.10) was used to convert the text pitch data into numerical values. MatPlotLib (https://matplotlib.org/ version 3.9.0) was used to visualize the learning curves for the KNN model. Lastly, built in Python packages collections, os, and random were used to create dictionaries, get file paths, and generate random notes respectively.

3.1 Data Collection and Pre-Processing

The song data used in this project was collected from the Suzuki Cello School books, volumes 2 and 3 [12]. The first step of acquiring the data was to digitize the music, which was done by scanning the songs into a PDF format. Once all the music was converted Soundslice, (https://www.soundslice.com/) an online tool [13], was used to convert the PDFs to music XML files (see image below for example). Finally, these files could be used to load the music data into Python for use in the machine learning models. For each song, the MIDI pitch, duration, offset within a measure, and fingerings were extracted from the music XML to be used as training features and target.



```xml
<?xml version='1.0' encoding='UTF-8'?>
<!DOCTYPE score-partwise PUBLIC "-//Recordare//DTD MusicXML 3.0 Partwise//EN" "http://www.musicxml.org/dtds/partwise.dtd">
<score-partwise version="4.0">
  <identification>
    <encoding>
      <supports element="accidental" type="yes"/>
      <supports element="transpose" type="yes"/>
      <software>Soundslice MusicXML exporter</software>
      <encoding-date>2025-02-16</encoding-date>
    </encoding>
  </identification>
  <part-list>
    <score-part id="P1">
      <part-name>Track 1</part-name>
    </score-part>
  </part-list>
  <part id="P1">
    <measure number="1">
      <attributes>
        <divisions>8</divisions>
        <key>
          <fifths>0</fifths>
          <mode>major</mode>
        </key>
        <time symbol="common">
          <beats>4</beats>
          <beat-type>4</beat-type>
        </time>
        <clef>
          <sign>F</sign>
          <line>4</line>
        </clef>
      </attributes>
      <sound tempo="120"/>
      <direction>
        <direction-type>
          <dynamics>
            <mf/>
          </dynamics>
        </direction-type>
      </direction>
      <note>
        <pitch>
          <step>C</step>
          <octave>3</octave>
        </pitch>
        <duration>8</duration>
        <type>quarter</type>
        <stem>up</stem>
        <notations>
          <technical>
            <fingering placement="above">4</fingering>
          </technical>
        </notations>
      </note>
```

Example of music XML format

3.2 BiLSTM Full Sequence Data

The initial model developed in this study employs a sequence-to-sequence learning framework designed to accept full songs as a single input. The dataset was organized such that each song's notes formed one input sequence with a corresponding target sequence of fingerings. Sequences were padded to the maximum length across the dataset to enable batch processing. To maintain sequence integrity during evaluation, a group-based shuffle split was employed, ensuring entire songs were assigned to either the training or testing set without overlap. The neural architecture for the first model consisted of two stacked bidirectional Long Short-Term Memory (BiLSTM) layers, the first with 64 units and the second with 32 units, both configured to return full sequences to preserve alignment. A final dense layer with a softmax activation function projected the LSTM outputs into five classes representing possible finger choices. The model was trained using the Adam optimizer with a sparse categorical cross-entropy loss function and evaluated using accuracy. This architecture was selected to allow the model to capture both forward and backward dependencies in musical phrasing and fingering context.

Pseudocode:

```
import_music()
- Load all MusicXML files from two folders [4]
- For each note in each song:
   - Extract pitch (convert to MIDI), duration, offset, fingering [4]
- Store everything in a DataFrame and return it [5]

create_fake_data(df, num_songs)
- For each pitch in the dataset, compute fingering distribution [6]
- For each fake song:
   - Generate 300 notes by:
      - Randomly selecting pitch and duration [6]
      - Adjusting offset to wrap around every 4 beats
      - Selecting fingering based on learned distribution
   - Append these fake notes to the DataFrame [5]
- Return the new DataFrame with fake songs added

calculate_fingering_distribution(pitch, df)
- Filter the DataFrame for the given pitch
- Count how often each fingering appears
- Return normalized distribution (probabilities)

choose_fingering(fingering_distribution)
- Randomly choose a fingering based on probability distribution [6]
```

pitch_to_midi(pitch)
- Convert pitch name (e.g., "C#4") to MIDI number [9]
- Return 0 if pitch is invalid or unrecognized

split_data(df)
- Group the data by song
- Convert each song into a sequence of features and labels
- Pad sequences to the same length [7]
- Identify fake vs real songs
- Split real songs into train/test sets (80/20) [8]
- Include all fake songs in training set [6]
- Return train and test arrays for both input (X) and output (Y)

train_test_model(x_train, x_test, y_train, y_test)
- Build a Keras Sequential model: [7]
    - Two BiLSTM layers
    - One Dense softmax layer (5 units)
- Compile and train the model on training data
- Evaluate it on test data
- Return loss and accuracy

main()
- Load and process music data
- Optionally augment with fake data
- Split into training and test sets
- Train and evaluate the model
- Print loss and accuracy

3.3 BiLSTM Sliding Window

The second model adopts a localized sliding window approach to fingering prediction, aimed at leveraging a fixed-size context around each target note. Rather than treating entire songs as single sequences, the model constructs overlapping input windows of five consecutive notes, centered on the note whose fingering is to be predicted. Each window includes pitch, duration, and offset features for the five notes, resulting in a structured input of shape (5, 3) per sample. This approach captures immediate context while enabling the model to generalize across different pieces more effectively. The neural network used for this model consists of two stacked BiLSTM layers, the first with 64 units followed by a second with 32 units and configured to condense the context into a single representation. This is followed by a dense output layer with five units and a softmax activation function, corresponding to the possible fingering classes. The model is trained using the Adam optimizer and sparse categorical cross-entropy loss, and performance is evaluated based on classification accuracy. This architecture is designed to exploit short-term dependencies in the note sequence while maintaining sufficient flexibility to handle varied musical passages.

Pseudocode:

```
import_music(return_dataframe=False)
- Load all MusicXML files from two folders [4]
- For each note in each song:
    - Extract pitch (convert to MIDI), duration, offset, fingering [4]
- Store this info:
    - In a DataFrame (if requested) [5]
    - As a structured list of song dictionaries
- Return songs list (and optionally the DataFrame too)

create_fake_data(df, num_songs)
- For each pitch in the dataset compute fingering distribution [6]
- For each fake song:
    - Generate 300 notes with:
        - Random pitch and duration [6]
        - Adjust offset to wrap around every 4 beats
        - Selecting fingering based on learned distribution
    - Append the new song to the fake songs list [5]
- Return list of fake song dictionaries

calculate_fingering_distribution(pitch, df)
- Filter the DataFrame for the given pitch
- Count how often each fingering appears
- Return normalized distribution (probabilities)

choose_fingering(fingering_distribution)
- Randomly choose a fingering based on probability distribution [6]

pitch_to_midi(pitch)
- Convert pitch name (e.g., "C#4") to MIDI number [9]
- Return 0 if pitch is invalid or unrecognized

create_sliding_windows(notes, window_size=2, include_features=False)
- For a sequence of notes:
    - Create windows of notes centered on each note (e.g. 2 before, 2 after)
    - If include_features:
        - Each input is a list of pitch, duration, offset
    - Else:
        - Each input is just pitch
    - Output is the fingering of the center note
- Return arrays of input windows and output fingerings [6]
```

prepare_dataset(songs, include_features=False)

```
- For each song in the dataset:
    - Create sliding windows and targets
- Stack all windows into final X and Y arrays
- Return input and target arrays [6]
```

train_test_model(x_train, x_test, y_train, y_test)

```
- Build a Keras Sequential model: [7]
    - Two BiLSTM layers
    - One Dense softmax layer (5 units)
- Compile and train the model on training data
- Evaluate it on test data
- Return loss and accuracy
```

main()

```
- Load and process music data
- Optionally augment with fake data
- Split into training and test sets [8]
- Train and evaluate the model
- Print loss and accuracy
```

3.4 Single Note Predictions

The third model evaluated in this study employs a simplified, non-sequential approach, predicting the fingering for each note based solely on its pitch. This design serves as a baseline model to assess the predictive power of individual pitch values in the absence of contextual information. The dataset is split randomly at the individual note level using a standard 80/20 train-test split, without preserving song boundaries. The resulting dataset consists of flat, unstructured vectors representing single pitches as inputs. The model itself is a fully connected feedforward neural network composed of three dense layers with 128, 64, and 32 units respectively, each using ReLU activation. A final softmax output layer maps to five possible fingering classes. The model is compiled using the Adam optimizer and trained with sparse categorical cross-entropy loss. Training is conducted over 2000 epochs, and evaluation is based on classification accuracy. This architecture provides a straightforward benchmark to compare against more context-aware models and helps isolate the effectiveness of pitch alone as a predictor for fingering choice.

Pseudocode:

```
import_music()
- Load all MusicXML files from two folders [4]
- For each note in each song:
    - Extract pitch (convert to MIDI), duration, offset, fingering [4]
- Combine all notes into a flat list (not grouped by song)
- Return list of notes

create_fake_data(notes, num_notes)
- Count how often each fingering appears per pitch
- Convert counts into fingering probability distributions
- Generate ‘num_notes’ synthetic notes by:
    - Call generate_new_pitch_fingering to get random pitch and associated fingering
    - Add new note to fake notes list
- Return list of fake notes

generate_new_pitch_fingering(pitch_fingering_probs)
- Randomly choose a pitch from the dataset
- Choose a fingering for that pitch based on its probability distribution
- Return a dictionary with pitch and fingering

pitch_to_midi(pitch)
- Convert pitch name (e.g., "C#4") to MIDI number [9]
- Return 0 if pitch is invalid or unrecognized

prepare_dataset(notes)
- Convert a list of notes into two arrays:
    - ‘x’: list of pitches (features)
    - ‘y’: list of fingerings (labels)
- Return arrays suitable for training [6]

train_test_model(x_train, x_test, y_train, y_test)
- Build a dense feedforward neural network: [7]
    - 3 hidden layers with ReLU activation
    - 1 softmax output layer (5 units)
- Compile and train the model
- Evaluate on test data
- Return loss and accuracy
```

```
main()
- Load music data as a flat list of notes
- Split into training and testing sets [8]
- Convert to feature/label arrays
- Optionally augment with fake data [6]
- Train and evaluate the model
- Print loss and accuracy
```

3.5 KNN

Next a K-Nearest Neighbors (KNN) classifier was implemented to evaluate the performance of a simple distance-based method for predicting cello fingerings. The model takes note pitch feature vectors as input and assigns fingering labels based on the majority vote of the k nearest neighbors in the training set. The number of neighbors (k) was treated as a hyperparameter and varied from 1 to 10 to observe its effect on both training and testing accuracy. For each value of k, the model was trained and evaluated using accuracy as the performance metric, and the results were visualized in a learning curve. This approach provides insight into the bias-variance trade-off inherent in the KNN method, with smaller k values potentially overfitting and larger k values promoting generalization. Unlike deep learning architectures, KNN relies solely on geometric proximity in the feature space, making it highly interpretable but potentially limited in capturing complex contextual patterns. Nevertheless, it serves as an effective benchmark for assessing the value of more sophisticated modeling strategies.

Pseudocode:

```
import_music()
- Load all MusicXML files from two folders [4]
- For each note in each song:
    - Extract pitch (convert to MIDI), duration, offset, fingering [4]
- Combine all notes into a flat list (not grouped by song)
- Return list of notes

create_fake_data(notes, num_notes)
- Count how often each fingering appears per pitch
- Convert counts into fingering probability distributions
- Generate ‘num_notes’ synthetic notes by:
    - Call generate_new_pitch_fingering to get random pitch and associated fingering
    - Add new note to fake notes list
```

- Return list of fake notes

generate_new_pitch_fingering(pitch_fingering_probs)
- Randomly choose a pitch from the dataset
- Choose a fingering for that pitch based on its probability distribution
- Return a dictionary with pitch and fingering

pitch_to_midi(pitch)
- Convert pitch name (e.g., "C#4") to MIDI number [9]
- Return 0 if pitch is invalid or unrecognized

prepare_dataset(notes)
- Convert a list of notes into two arrays:
  - ‘x’: list of pitches (features)
  - ‘y’: list of fingerings (labels)
- Return arrays suitable for training [6]

train_test_model(x_train, x_test, y_train, y_test)
- For k = 1 to 9:
  - Create a KNN with k neighbors [8]
  - Train it on the training data
  - Evaluate and record training and test accuracy [8]
- Plot the accuracy vs. k for both training and testing sets [10]
- Show the plot (KNN learning curve for fingering classification) [10]

main()
- Load music data as a flat list of notes
- Split into training and testing sets [8]
- Convert to feature/label arrays
- Optionally augment with fake data [6]
- Train and evaluate the KNN by calling train_test_model

3.6 Transformer

The final model explored in this study employs a Transformer-based architecture to predict cello fingerings, leveraging attention mechanisms to capture contextual relationships within a localized window of notes. Each input sample consists of a sliding window of five consecutive notes, with each note represented by three features: pitch, note duration, and offset. These sequences are constructed from full musical pieces and organized into overlapping windows to serve as training samples. The model uses three separate input streams for pitch, duration, and offset, which are transformed into a shared embedding space. Pitch is embedded via a learned embedding layer, while duration and offset are passed through dense projection layers after being expanded to match the embedding dimensionality. The resulting embeddings are combined through element-wise addition and passed through two stacked Transformer encoder blocks. Each encoder consists of multi-head self-attention, dropout, residual connections, and feedforward sublayers with layer normalization, allowing the model to learn complex dependencies between notes in the window without relying on sequential recurrence. The output is passed through a dense layer with softmax activation to produce per-note fingering predictions across five classes. The model is trained using the Adam optimizer and sparse categorical cross-entropy loss, with accuracy as the evaluation metric. This architecture is designed to capture nuanced interdependencies among multiple note attributes within short musical phrases, offering a flexible and scalable alternative to traditional recurrent models.

Pseudocode:

```
import_music()
- Load all MusicXML files from two folders [4]
- For each note in each song:
   - Extract pitch (convert to MIDI), duration, offset, fingering [4]
- Store the data as a structured list of song dictionaries
- Return songs list

pitch_to_midi(pitch)
- Convert pitch name (e.g., "C#4") to MIDI number [9]
- Return 0 if pitch is invalid or unrecognized

transformer_encoder(inputs, head_size=64, num_heads=2, ff_dim=128, dropout=0.1)
- Apply multi-head self-attention to the inputs [7]
- Add dropout layer [7]
- Add residual connection and normalization [7]
- Pass result through feedforward dense layers with dropout [7]
- Add another residual connection and normalization [7]
- Return the transformed representation

create_model(seq_length, pitch_vocab_size, embedding_dim)
- Define input layers for pitch, duration, and offset [7]
- Embed pitch and project duration/offset to the same embedding dimension [7]
- Add the three embeddings together [7]
- Pass the result through two transformer encoder blocks
- Use a dense softmax layer to predict fingering for each note [7]
- Compile and return the model [7]
```

```
create_sliding_windows(data, seq_length)
- For each song:
    - Create overlapping windows of ‘seq_length’ notes
    - For each window:
        - Separate features (pitch, duration, offset)
        - Store features and corresponding fingerings as label
- Return arrays of inputs and labels [6]

train_and_test(x_pitch, x_duration, x_offset, y, seq_length, pitch_vocab_size,
embedding_dim)
- Perform 5-fold cross-validation: [8]
    - For each fold:
        - Split pitch, duration, offset, and fingering into train/test sets
        - Create and train a new transformer model on the training set
        - Evaluate the model on the test set
        - Record the test accuracy
        - Print accuracy for fold
- Average accuracy across all folds and print

main()
- Set transformer model hyperparameters:
    - Window size = 5
    - Pitch vocabulary = 128
    - Embedding size = 32
- Load music data (a list of songs, each as note sequences)
- Create sliding windows from notes
- Separate pitch, duration, offset, and fingering from each window
- Train and evaluate the model using cross-validation
```

3.7 Data Augmentation

Lastly, to explore whether additional training data could enhance model accuracy, a data augmentation strategy was implemented. This involved analyzing the original dataset to compute the probability distribution of fingerings for each unique pitch across all songs. Using these distributions, synthetic songs were generated by randomly sampling notes. Each generated note was assigned a random duration and offset that reset once each measure was filled, while the fingering was selected according to its observed frequency in the original data. For instance, if a particular pitch was played with the fourth finger 90% of the time, newly generated notes with that pitch had a 90% likelihood of being assigned the fourth finger. This approach aimed at producing realistic, yet diverse training examples grounded in the statistical patterns of actual cello fingering usage. The first two models (both BiLSTM based) augmented the original data by adding additional songs to the dataset. The next two models (dense feed-forward and KNN) augmented the original data by adding some number of additional individual notes to the dataset since they did not keep the notes organized by song as it was unnecessary for the given approach. Lastly, no data augmentation was used for the final transformer model since it was performing well without any additional data and the previous models did not show improvement when using the generated data.

4 Results

Each of the models were tested three times, while changing the number of features, data augmentation, and number of epochs. Accuracy on a held-out test set was used as the primary evaluation metric. The results of these tests can be seen in table 1 and figure 1 below.

Method	**Features**	**Data augmentation?**	**Epochs**	**Accuracy**
Full sequence	Pitch, duration, offset	No	10	0.662
Full sequence	Pitch, duration, offset	No	1000	0.678
Full sequence	Pitch, duration, offset	Yes, 30 songs	10	0.657
Sliding window	Pitch	No	1000	0.804
Sliding window	Pitch	Yes, 10 songs	1000	0.839
Sliding window	Pitch, duration, offset	No	1000	0.686
Single Datapoints	Pitch	No	1000	0.758
Single Datapoints	Pitch	No	2000	0.78
Single Datapoints	Pitch	Yes, 1000 notes	1000	0.282
Knn	Pitch	No	N/A	See Fig. 1
Knn	Pitch	Yes, 500 notes	N/A	See Fig. 2
Knn	Pitch	Yes, 1000 notes	N/A	See Fig. 3
Transformer	Pitch, duration, offset	No	50	0.933
Transformer	Pitch, duration, offset	No	100	0.942
Transformer	Pitch, duration, offset	No	200	0.954

Table 1

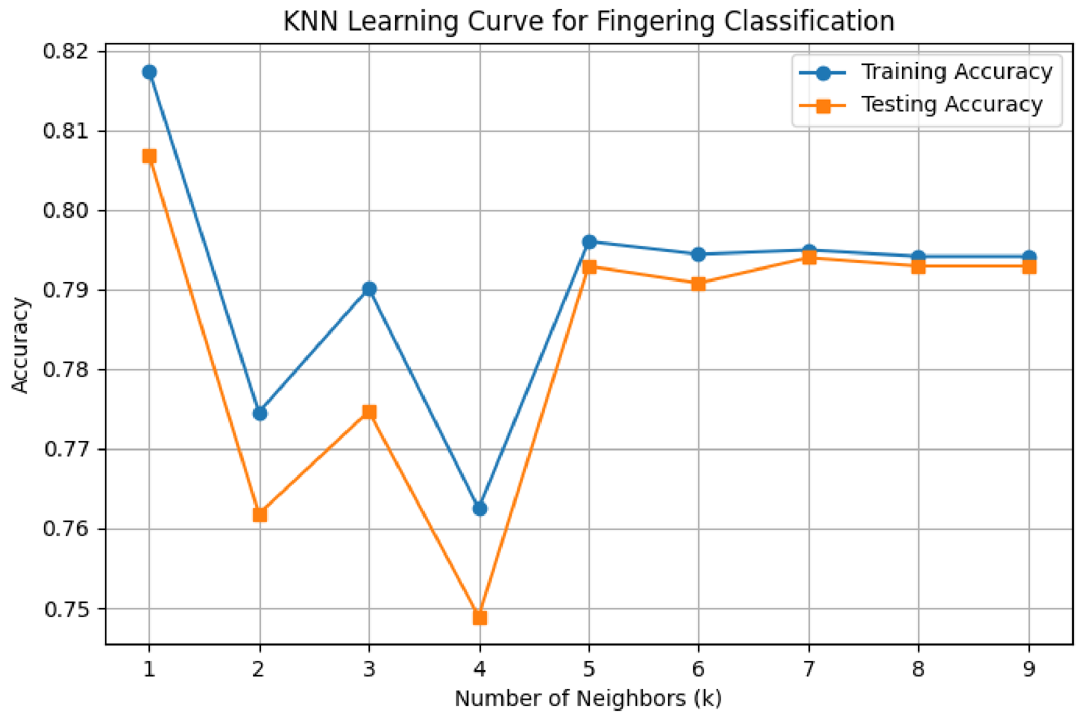

KNN Learning Curve for Fingering Classification
Training Accuracy
Testing Accuracy
Accuracy
0.82
0.81
0.80
0.79
0.78
0.77
0.76
0.75
1
2
3
4
5
6
7
8
9
Number of Neighbors (k)

Fig. 1

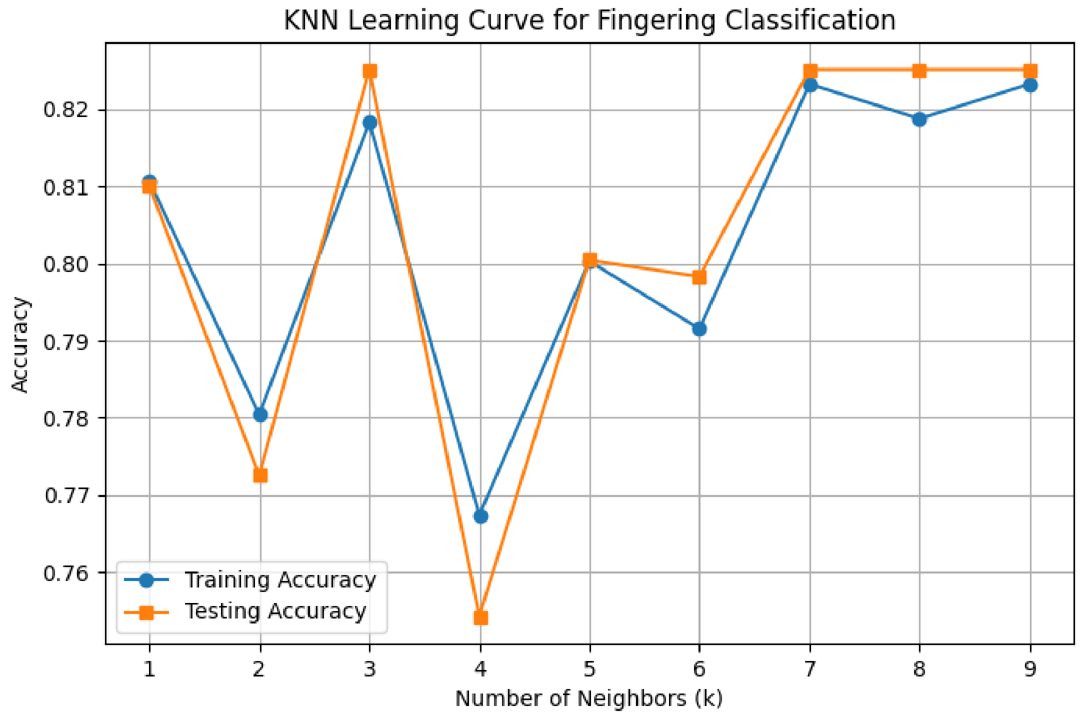

KNN Learning Curve for Fingering Classification
Training Accuracy
Testing Accuracy
Accuracy
0.82
0.81
0.80
0.79
0.78
0.77
0.76
1
2
3
4
5
6
7
8
9
Number of Neighbors (k)

Fig. 2

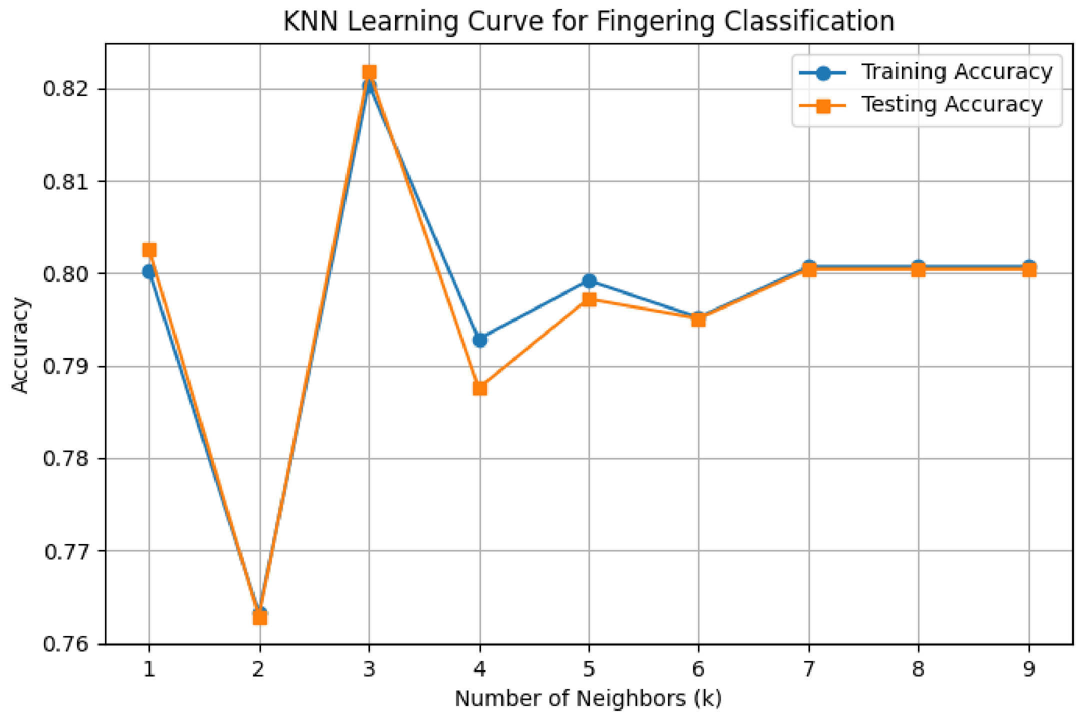


Fig. 3

Among all methods, the Transformer-based model consistently achieved the highest performance, with accuracy reaching 95.4% after 200 epochs. This suggests that attention-based architectures are especially effective at capturing the local and global dependencies among pitch, duration, and offset features in musical sequences. Even after only 50 epochs, the Transformer model surpassed all other methods, highlighting its efficiency in learning from structured musical data. The same accuracy was also achieved after performing 5-fold cross-validation to ensure the first data split was not skewed to allow for higher accuracy.

The sliding window BiLSTM models also performed well. Using pitch alone, the model achieved an accuracy of 80.4%, which increased to 83.9% when modest data augmentation (10 synthetic songs) was applied. Interestingly, incorporating duration and offset into the sliding window reduced performance (68.6%), indicating that either these features were not effectively utilized in this setup or that the increased input complexity introduced noise rather than benefit.

The full-sequence BiLSTM model, which treats each song as a complete sequence, achieved lower performance overall (max 67.8%), and showed little benefit from increased training epochs or the addition of synthetic data. This may suggest that full-song context is less useful for beginner-level music, where fingering decisions tend to depend more on local context.

The simpler pitch-only feedforward model (Single Datapoints) reached an accuracy of 78% after 2000 epochs, demonstrating that pitch alone provides a fairly strong signal for fingering prediction. However, when this model was trained with 1000 synthetic notes, performance dropped drastically to 28.2%, suggesting that the generated data may have introduced noisy or inconsistent patterns when not contextualized within sequences.

The K-Nearest Neighbors (KNN) classifier performed competitively, achieving a peak accuracy of just under 82% with $k = 1$. Accuracy declined with higher values of k, stabilizing around 79.5% for $k = 5$–9. With the addition of synthetic data, the KNN was able to achieve a slightly higher accuracy of just over 82% for 500 additional notes. This result indicates that KNN is effective for this task when tuned correctly, although it lacks the scalability and expressive power of the neural models.

Overall, the results demonstrate that local context (as modeled by the sliding window and Transformer approaches) is more predictive for cello fingering than full-sequence modeling, and that attention-based architectures offer substantial benefits when incorporating multiple note-level features.

5 Discussion

The findings of this study suggest that the choice of model architecture and input representation plays a crucial role in the effectiveness of automated cello fingering prediction. Notably, the Transformer-based model substantially outperformed all other approaches, indicating that attention mechanisms are particularly well-suited to capturing the complex interplay of pitch, timing, and duration that informs fingering decisions. Unlike recurrent architectures, the Transformer can process all elements in a window simultaneously, allowing it to learn flexible relationships across the input features without assuming a strictly linear sequence dependency.

The strong performance of the sliding window BiLSTM model further supports the hypothesis that local musical context is more important than full-piece structure when predicting fingerings, especially in beginner-level music. These models benefit from seeing a few notes before and after the target note, mimicking the way a human might consider nearby context to determine finger placement. However, the drop in performance when including duration and offset in the sliding window suggests that these features may not contribute meaningfully without more specialized preprocessing or attention-based weighting.

The relatively modest accuracy of the full-sequence BiLSTM model points to potential drawbacks in modeling long sequences in this domain. Beginner music often consists of short, repetitive patterns, making full-sequence modeling unnecessarily complex and possibly prone to overfitting or underutilizing long-range context. Additionally, the poor generalization of the pitch-only model when trained with synthetic data highlights the importance of musical coherence—randomly generated notes may lack the structural and temporal consistency necessary for effective learning, especially in models that do not incorporate contextual constraints.

Surprisingly, the KNN classifier achieved respectable performance, rivaling that of more complex models in some configurations. This outcome emphasizes the strength of pitch as a predictive feature on its own and suggests that relatively simple models can serve as effective baselines in music learning applications.

Overall, these results reinforce the value of modeling short-term musical context and incorporating structured input features using architectures designed to capture interdependent relationships. The Transformer model's success indicates promising potential for real-world implementation in intelligent tutoring systems or digital music education tools, particularly for beginner cellists who benefit from clear, consistent fingering guidance.

Works Cited

[1] W. Gao, S. Zhang, N. Zhang, X. Xiong, Z. Shi, and K. Sun, "Generating Fingerings for Piano Music with Model-Based Reinforcement Learning" 2023 Applied Sciences 13, no. 20: 11321. https://doi.org/10.3390/app132011321

[2] A. P. Iman and C. W. Ahn, "A Model-Free Deep Reinforcement Learning Approach to Piano Fingering Generation," 2024 IEEE Conference on Artificial Intelligence (CAI), Singapore, Singapore, 2024, pp. 31-37, doi: 10.1109/CAI59869.2024.00016.

[3] Vincent K.M. Cheung, Hsuan-Kai Kaoand Li Su, "Semi-supervised violin fingering generation using variational autoencoders", in Proceedings of the 22nd International Society for Music Information Retrieval Conference, Online, Nov. 2021, pp. 113–120. doi: 10.5281/zenodo.5624441.

[4] Cuthbert, M. S., & Ariza, C. (2010). *music21: A Toolkit for Computer-Aided Musicology*. In *Proceedings of the International Society for Music Information Retrieval Conference* (ISMIR), 2010. https://web.mit.edu/music21 (Version 9.5.0)

[5] The pandas development team. (2020). *pandas-dev/pandas: Pandas* (Version 2.2.3) [Computer software]. Zenodo. https://doi.org/10.5281/zenodo.3509134

[6] Harris, C.R., Millman, K.J., van der Walt, S.J. et al. *Array programming with NumPy*. Nature 585, 357–362 (2020). DOI: 10.1038/s41586-020-2649-2. (Publisher link). (Version 1.26.4)

[7] Chollet, F., & others. (2015). *Keras* [Computer software]. https://keras.io (Version 3.8.0)

[8] Pedregosa, F., Varoquaux, Ga"el, Gramfort, A., Michel, V., Thirion, B., Grisel, O., … others. (2011). Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, *12*(Oct), 2825–2830. (Version 1.6.1)

[9] Raffel, C., & Ellis, D. P. W. (2014). *Intuitive Analysis, Creation and Manipulation of MIDI Data with pretty_midi*. In *Proceedings of the 15th International Conference on Music Information Retrieval* (ISMIR), 2014. https://colinraffel.com/publications/ismir2014_prettymidi.pdf (Version 0.2.10)

[10] J. D. Hunter, *Matplotlib: A 2D Graphics Environment* in *Computing in Science & Engineering*, vol. 9, no. 3, pp. 90-95, May-June 2007, doi: 10.1109/MCSE.2007.55. (Version 3.9.0)

[11] Python Software Foundation. (2023). *Python (Version 3.12)* [Computer software]. https://www.python.org

[12] Suzuki, Shinichi. *Suzuki Cello School, Volume 2*. Revised ed. Alfred Music, 1999.

[13] *Create Living Sheet Music*. *Soundslice*, www.soundslice.com/. Accessed 16 Feb. 2025.