NFLWin Documentation

Release 1.0.0

Andrew Schechtman-Rook

Aug 17, 2016

Links

1 Quickstart **3**

2 Current Default Model **5**

3 Why NFLWin? **7**

4 Resources **9**
- 4.1 Installation . . . 9
- 4.2 Creating a New WP Model . . . 10
- 4.3 Using Data From nfldb . . . 12
- 4.4 For Developers . . . 15
- 4.5 nflwin . . . 17

Python Module Index **29**

NFLWin is designed from the ground up to provide two things:

- A simple-to-use interface for users to compute Win Probabilities (WP) for NFL plays based on a built-in WP model.
- A robust framework for improving estimates of WP.

NFLWin builds on scikit-learn's `fit-transform` idiom, allowing for pipelines that take in raw box score data and return estimated WPs - all data preprocessing takes place behind the scenes. Additionally, these preprocessing steps can be easily reordered, replaced, and/or extended, allowing for rapid iteration and prototyping of potential improvements to the WP model.

NFLWin also has built-in support for efficiently querying data from nfldb directly into a format useable by the built-in WP model, although the model is fully data-source-agnostic as long as the data is formatted properly for the model to parse.

CHAPTER 1

Quickstart

NFLWin is `pip`-installable:

```
$ pip install nflwin
```

Note: NFLWin depends on SciPy, which is notoriously difficult to install properly via `pip`. You may wish to use the Conda package manager to install Scipy before installing NFLWin.

When installed via `pip`, NFLWin comes with a working Win Probability model out-of-the-box:

```
>>> from nflwin.model import WPModel
>>> standard_model = WPModel.load_model()
```

The default model can be inspected to learn what data it requires:

```
>>> standard_model.column_descriptions
{'home_team': 'Abbreviation for the home team', 'yardline': "The yardline, given by (yards from own g
```

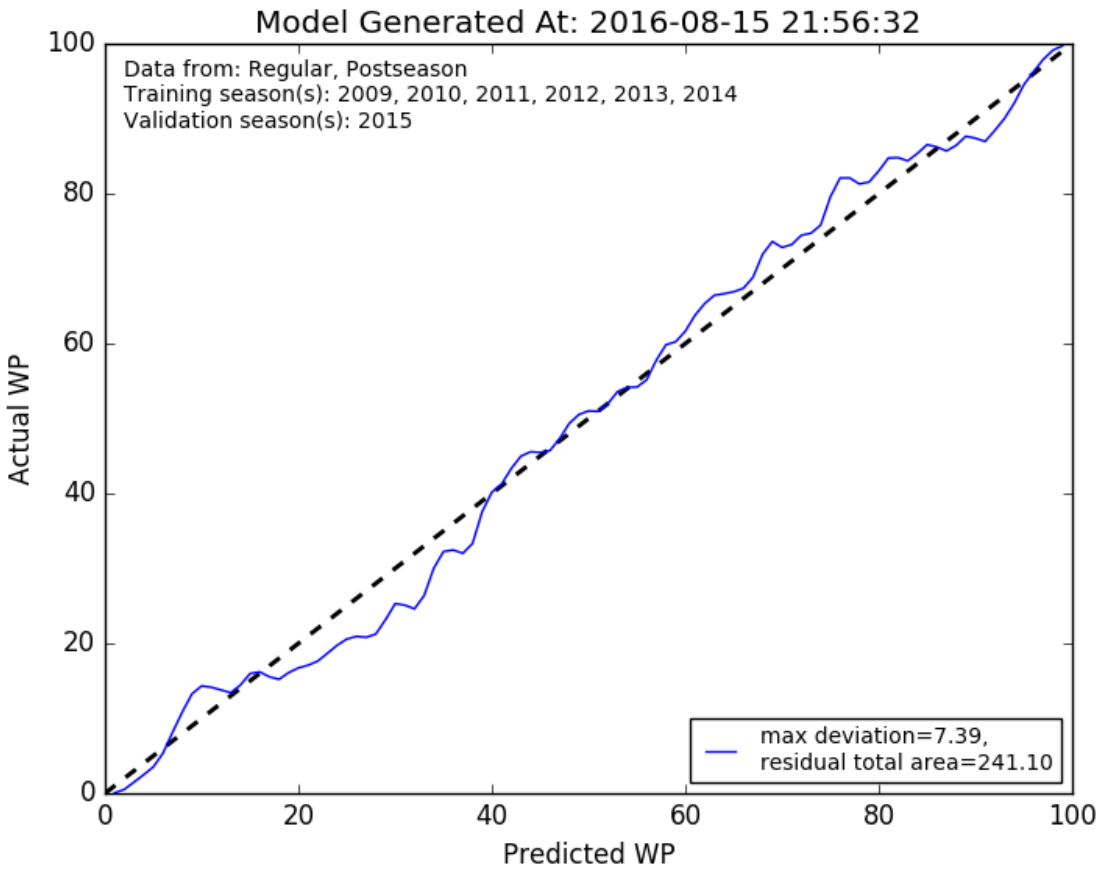
NFLWin operates on Pandas DataFrames:

```
>>> import pandas as pd
>>> plays = pd.DataFrame({
... "quarter": ["Q1", "Q2", "Q4"],
... "seconds_elapsed": [0, 0, 600],
... "offense_team": ["NYJ", "NYJ", "NE"],
... "yardline": [-20, 20, 35],
... "down": [1, 3, 3],
... "yards_to_go": [10, 2, 10],
... "home_team": ["NYJ", "NYJ", "NYJ"],
... "away_team": ["NE", "NE", "NE"],
... "curr_home_score": [0, 0, 21],
... "curr_away_score": [0, 0, 10]
... })
```

Once data is loaded, using the model to predict WP is easy:

```
>>> standard_model.predict_wp(plays)
array([ 0.58300397,  0.64321796,  0.18195466])
```

Current Default Model



CHAPTER 3

Why NFLWin?

Put simply, there are no other options: while WP models have been widely used in NFL analytics for years, the analytics community has almost totally dropped the ball in making these models available for the general public or even explaining their algorithms at all.

For a (much) longer explanation, see the PhD Football blog.

CHAPTER 4

Resources

4.1 Installation

NFLWin only supports Python 2, as nfldb is currently incompatible with Python 3. The bulk of NFLWin should work natively with Python 3, however that is currently untested. Pull requests ensuring this compatibility would be welcome.

4.1.1 Releases

Stable releases of NFLWin are available on PyPI:

```
$ pip install nflwin
```

The default install provides exactly the tools necessary to make predictions using the standard WP model as well as make new models. However it does not include the dependencies necessary for *using nfldb*, producing diagnostic plots, or contributing to the package.

Installing NFLWin with those extra dependencies is accomplished by adding a parameter in square brackets:

```
$ pip install nflwin[plotting] #Adds matplotlib for plotting
$ pip install nflwin[nfldb] #Dependencies for using nfldb
$ pip install nflwin[dev] #Everything you need to develop on NFLWin
```

Note: NFLWin depends on the scipy library, which is notoriously difficult to install via pip or from source. One option if you're having difficulty getting scipy installed is to use the Conda package manager. After installing Conda, you can create a new environment and install dependencies manually before pip installing NFLWin:

```
$ conda create -n nflwin-env python=2.7 numpy scipy scikit-learn pandas
```

4.1.2 Bleeding Edge

If you want the most recent stable version you can install directly from GitHub:

```
$ pip install git+https://github.com/AndrewRook/NFLWin.git@master#egg=nflwin
```

You can append the arguments for the extra dependencies in the same way as for the installation from PyPI.

Note: GitHub installs **do not** come with the default model. If you want to use a GitHub install with the default model, you'll need to install NFLWin from PyPI somewhere else and then copy the model into the model directory from your GitHub install. If you need to figure out where that directory is, print `model.WPModel.model_directory`.

4.2 Creating a New WP Model

While NFLWin ships with a fairly robust default model, there is always room for improvement. Maybe there's a new dataset you want to use to train the model, a new feature you want to add, or a new machine learning model you want to evaluate.

Good news! NFLWin makes it easy to train a new model, whether you just want to refresh the data or to do an entire refit from scratch. We'll start with the simplest case:

4.2.1 Default Model, New Data

Refreshing the data with NFLWin is a snap. If you want to change the data used by the default model but keep the source as nfldb, all you have to do is override the default keyword arguments when calling the *train_model()* and *validate_model()* methods. For instance, if for some insane reason you wanted to train on the 2009 and 2010 regular seasons and validate on the 2011 and 2012 playoffs, you would do the following:

```
>>> from nflwin.model import WPModel
>>> new_data_model = WPModel()
>>> new_data_model.train_model(training_seasons=[2009, 2010], training_season_types=["Regular"])
>>> new_data_model.validate_model(validation_seasons=[2011, 2012], validation_season_types=["Postseas
(21.355462918011327, 565.56909036318007)
```

If you want to supply your own data, that's easy too - simply set the *source_data* kwarg of *train_model()* and *validate_model()* to be a Pandas DataFrame of your training and validation data (respectively):

```
>>> from nflwin.model import WPModel
>>> new_data_model = WPModel()
>>> training_data.head()
      gsis_id  drive_id  play_id offense_team  yardline  down  yards_to_go  \
0  2012090500         1       35          DAL     -15.0     0            0
1  2012090500         1       57          NYG     -34.0     1           10
2  2012090500         1       79          NYG     -34.0     2           10
3  2012090500         1      103          NYG     -29.0     3            5
4  2012090500         1      125          NYG     -29.0     4            5

  home_team away_team offense_won quarter  seconds_elapsed  curr_home_score  \
0       NYG       DAL         True      Q1              0.0                0
1       NYG       DAL        False      Q1              4.0                0
2       NYG       DAL        False      Q1             11.0                0
3       NYG       DAL        False      Q1             55.0                0
4       NYG       DAL        False      Q1             62.0                0

   curr_away_score
0                0
1                0
2                0
3                0
4                0
>>> new_data_model.train_model(source_data=training_data)
```

```
>>> validation_data.head()
      gsis_id  drive_id  play_id offense_team  yardline  down  yards_to_go  \
0  2014090400         1       36          SEA     -15.0     0            0
1  2014090400         1       58           GB     -37.0     1           10
2  2014090400         1       79           GB     -31.0     2            4
3  2014090400         1      111           GB     -26.0     1           10
4  2014090400         1      132           GB     -11.0     1           10

  home_team away_team offense_won quarter  seconds_elapsed  curr_home_score  \
0       SEA        GB        True      Q1              0.0                0
1       SEA        GB       False      Q1              4.0                0
2       SEA        GB       False      Q1             30.0                0
3       SEA        GB       False      Q1             49.0                0
4       SEA        GB       False      Q1             88.0                0

   curr_away_score
0                0
1                0
2                0
3                0
4                0
>>> new_data_model.validate_model(source_data=validation_data)
(8.9344062502671591, 265.7971863696315)
```

4.2.2 Building a New Model

If you want to construct a totally new model, that's possible too. Just instantiate *WPModel*, then replace the `model` attribute with either a scikit-learn classifier or Pipeline. From that point *train_model()* and *validate_model()* should work as normal.

Note: If you create your own model, the `column_descriptions` attribute will no longer be accurate unless you update it manually.

Note: If your model uses a data structure other than a Pandas DataFrame, you will not be able to use the `source_data="nfldb"` default kwarg of *train_model()* and *validate_model()*. If you want to use nfldb data, query it through *nflwin.utilities.get_nfldb_play_data()* first and convert it from a DataFrame to the format required by your model.

Using NFLWin's Preprocessors

While you can completely roll your own WP model from scratch, NFLWin comes with several classes designed to aid in preprocessing your data. These can be found in the appropriately named *preprocessing* module. Each of these preprocessors inherits from scikit-learn's BaseEstimator class, and therefore is fully compatible with scikit-learn Pipelines. Available preprocessors include:

- *ComputeElapsedTime*: Convert the time elapsed in a quarter into the total seconds elapsed in the game.
- *ComputeIfOffenseIsHome*: Create an indicator variable for whether or not the offense is the home team.
- *CreateScoreDifferential*: Create a column indicating the difference between the offense and defense point totals (offense-defense). Uses home team and away team plus an indicator giving if the offense is the home team to compute.

- *MapToInt*: Map a column of values to integers. Useful for string columns (e.g. a quarter column with "Q1", "Q2", etc).
- *CheckColumnNames*: Ensure that only the desired data gets passed to the model in the right order. Useful to guarantee that the underlying numpy arrays in a Pandas DataFrame used for model validation are in the same order as they were when the model was trained.

To see examples of these preprocessors in use to build a model, look at *nflwin.model.WPModel.create_default_pipeline()*.

4.2.3 Model I/O

To save a model to disk, use the *nflwin.model.WPModel.save_model()* method.

Note: If you do not provide a filename, the default model will be overwritten and in order to recover it you will need to reinstall NFLWin (which will then overwrite any non-default models you have saved).

To load a model from disk, use the *nflwin.model.WPModel.load_model()* class method. By default this will load the standard model that comes bundled with pip installs of NFLWin. Simply specify the `filename` kwarg to load a non-standard model.

Note: By default, models are saved to and loaded from the path given by *nflwin.model.WPModel.model_directory*, which by default is located inside your NFLWin install.

4.2.4 Estimating Quality of Fit

When you care about measuring the probability of a classification model rather than getting a yes/no prediction it is challenging to estimate its quality. This is an area I'm actively looking to improve upon, but for now NFLWin does the following.

First, it takes the probabilities given by the model for each play in the validation set, then produces a kernel density estimate (KDE) of all the plays as well as just the ones that were predicted correctly. The ratio of these two KDEs is the actual WP measured from the test data set at a given *predicted* WP. While all of this is measured in *validate_model()*, you can plot it for yourself by calling the *plot_validation()* method, which will generate a plot like that shown on the home page.

From there NFLWin computes both the maximum deviation at any given percentage and the total area between the estimated WP from the model and what would be expected if the model was perfect - that's what is actually returned by *validate_model()*. This is obviously not ideal given that it's not directly estimating uncertainties in the model, but it's the best I've been able to come up with so far. If anyone has an idea for how to do this better I would welcome it enthusiastically.

4.3 Using Data From nfldb

NFLWin comes with robust support for querying data from nfldb, a package designed to facilitate downloading and accessing play-by-play data. There are functions to query the nfldb database in *nflwin.utilities*, and *nflwin.model.WPModel* has keyword arguments that allow you to directly use nfldb data to fit and validate a WP model. Using nfldb is totally optional: a default model is already fit and ready to use, and NFLWin is fully compatible with any source for play-by-play data. However, nfldb is one of the few free sources of up-to-date NFL data and so it may be a useful resource to have.

4.3.1 Installing nfldb

nfldb is pip-installable, and can be installed as an extra dependency (`pip install nflwin[nfldb]`). Without setting up the nfldb Postgres database first, however, the pip install will succeed but nfldb will be unuseable. What's more, trying to set up the database *after* installing nfldb may fail as well.

The nfldb wiki has fairly decent installation instructions, but I know that when I went through the installation process I had to interpret and adjust several steps. I'd at least recommend reading through the wiki first, but in case it's useful I've listed the steps I followed below (for reference I was on Mac OS 10.10).

Installing Postgres

I had an old install kicking around, so I first had to clean that up. Since I was using Homebrew:

```
$ brew uninstall --force postgresql
$ rm -rf /usr/local/var/postgres/ # where I'd installed the prior DB
```

Then install a fresh version:

```
$ brew update
$ brew install postgresql
```

Start Postgres and Create a Default DB

You can choose to run Postgres at startup, but I don't use it that often so I choose not to do those steps - I just run it in the foreground with this command:

```
$ postgres -D /usr/local/var/postgres
```

Or in the background with this command:

```
$ pg_ctl -D /usr/local/var/postgres -l logfile start
```

If you don't create a default database based on your username, launching Postgres will fail with a `psql: FATAL: database "USERNAME" does not exist` error:

```
$ createdb `whoami`
```

Check that the install and configuration went well by launching Postgres as your default user:

```
$ psql
psql (9.5.2)
Type "help" for help.

USERNAME=#
```

Next, add a password:

```
USERNAME=# ALTER ROLE "USERNAME" WITH ENCRYPTED PASSWORD 'choose a
superuser password';
USERNAME=# \q;
```

Edit the `pg_hba.conf``file found in your database (in my case the file was ``/usr/local/var/postgres/pg_hba.conf`), and change all instances of `trust` to `md5`.

Create nfldb Postgres User and Database

Start by making a user:

```
$ createuser -U USERNAME -E -P nfldb
```

where you replace `USERNAME` with your actual username. Make up a new password. Then make the nfldb database:

```
$ createdb -U USERNAME -O nfldb nfldb
```

You'll need to enter the password for the USERNAME account. Next, add the fuzzy string matching extension:

```
$ psql -U USERNAME -c 'CREATE EXTENSION fuzzystrmatch;' nfldb
```

You should now be able to connect the nfldb user to the nfldb database:

```
$ psql -U nfldb nfldb
```

From this point you should be able to follow along with the instructions from nfldb.

4.3.2 Using nfldb

Once nfldb is properly installed, you can use it with NFLwin in a couple of different ways.

Querying Data

nfldb comes with a robust set of options to query its database, but they tend to be designed more for ad hoc querying of small amounts of data or computing aggregate statistics. It's possible to use built-in nfldb queries to get the data NFLWin needs, but it's *slow*. So NFLWin has built in support for bulk queries of nfldb in the *nflwin.utilities* module:

```
>>> from nflwin import utilities
>>> data = utilities.get_nfldb_play_data(season_years=[2010],
... season_types=["Regular", "Postseason"])
>>> data.head()
      gsis_id  drive_id  play_id offense_team  yardline  down  yards_to_go  \
0  2010090900         1       35          MIN     -20.0     0            0
1  2010090900         1       57           NO     -27.0     1           10
2  2010090900         1       81           NO       1.0     1           10
3  2010090900         1      109           NO      13.0     1           10
4  2010090900         1      135           NO      13.0     2           10

  home_team away_team offense_won quarter  seconds_elapsed  curr_home_score  \
0        NO       MIN       False      Q1              0.0                0
1        NO       MIN        True      Q1              4.0                0
2        NO       MIN        True      Q1             39.0                0
3        NO       MIN        True      Q1             79.0                0
4        NO       MIN        True      Q1             84.0                0

   curr_away_score
0                0
1                0
2                0
3                0
4                0
```

You can see the docstring for more details, but basically `get_nfldb_play_data` queries the nfldb database directly for columns relevant to estimating WP, does some simple parsing/preprocessing to get them in the right format, then returns them as a dataframe. Keyword arguments control what parts of seasons are queried.

Integration with WPModel

While you can train NFLWin's win probability model (*nflwin.model.WPModel*) with whatever data you want, it comes with keyword arguments that allow you to query nfldb directly. For instance, to train the default model on the 2009 and 2010 regular seasons from nfldb, you'd enter the following:

```
>>> from nflwin.model import WPModel
>>> model = WPModel()
>>> model.create_default_pipeline()
Pipeline(...)
>>> model.train_model(source_data="nfldb",
... training_seasons=[2009, 2010],
... training_season_types=["Regular"])
```

4.4 For Developers

This section of the documentation covers things that will be useful for those already contributing to NFLWin.

Note: Unless stated otherwise assume that all filepaths given in this section start at the root directory for the repo.

4.4.1 Testing Documentation

Documentation for NFLWin is hosted at Read the Docs, and is built automatically when changes are made on the master branch or a release is cut. However, oftentimes it's valuable to display NFLWin's documentation locally as you're writing. To do this, run the following:

```
$ ./build_local_documentation.sh
```

When that command finishes, open up `doc/index.html` in your browser of choice to see the site.

4.4.2 Updating the Default Model

NFLWin comes with a pre-trained model, but if the code generating that model is updated **the model itself is not**. So you have to update it yourself. The good news, however, is that there's a script for that:

```
$ python make_default_model.py
```

Note: This script hardcodes in the seasons to use for training and testing samples. After each season those will likely need to be updated to use the most up-to-date data.

Note: This script requires `matplotlib` in order to run, as it produces a validation plot for the documentation.

4.4.3 Cutting a New Release

NFLWin uses semantic versioning, which basically boils down to the following (taken directly from the webpage linked earlier in this sentence):

> Given a version number MAJOR.MINOR.PATCH, increment the:
>
> 1. MAJOR version when you make incompatible API changes,
> 2. MINOR version when you add functionality in a backwards-compatible manner, and
> 3. PATCH version when you make backwards-compatible bug fixes.

Basically, unless you change something drastic you leave the major version alone (the exception being going to version 1.0.0, which indicates the first release where the interface is considered "stable").

The trick here is to note that information about a new release must live in a few places:

- In `nflwin/_version.py` as the value of the `__version__` variable.
- As a tagged commit.
- As a release on GitHub.
- As an upload to PyPI.
- (If necessary) as a documented release on Read the Docs.

Changing the version in one place but not in others can have relatively minor but fairly annoying consequences. To help manage the release cutting process there is a shell script that automates significant parts of this process:

```
$ ./increment_version.sh [major|minor|patch]
```

This script does a bunch of things, namely:

1. Parse command line arguments to determine whether to increment major, minor, or patch version.
2. Makes sure it's not on the master branch.
3. Makes sure there aren't any changes that have been staged but not committed.
4. Makes sure there aren't any changes that have been committed but not pushed.
5. Makes sure all unit tests pass.
6. Compares current version in nflwin/_version.py to most recent git tag to make sure they're the same.
7. Figures out what the new version should be.
8. Updates nflwin/_version.py to the new version.
9. Uploads package to PyPI.
10. Adds and commits nflwin/_version.py with commit message "bumped [TYPE] version to [VERSION]", where [TYPE] is major, minor, or patch.
11. Tags latest commit with version number (no 'v').
12. Pushes commit and tag.

It will exit if **anything** returns with a non-zero exit status, and since it waits until the very end to upload anything to PyPI or GitHub if you do run into an error in most cases you can fix it and then just re-run the script.

The process for cutting a release is as follows:

1. Make double sure that you're on a branch that's not `master` and you're ready to cut a new release (general good practice is to branch off from master *just* for the purpose of making a new release).

2. Run the `increment_version.sh` script.
3. Fix any errors, then rerun the script until it passes.
4. Make a PR on GitHub into master, and merge it in (self-merge is ok if branch is just updating version).
5. Make release notes for new release on GitHub.
6. (If necessary) go to Read the Docs and activate the new release.

4.5 nflwin

4.5.1 nflwin package

Submodules

nflwin.model module

Tools for creating and running the model.

class `nflwin.model.`**`WPModel`**(*copy_data=True*)
Bases: `object`

The object that computes win probabilities.

In addition to holding the model itself, it defines some columns names likely to be used in the model as parameters to allow other users to more easily figure out which columns go into the model.

Parameters **copy_data** : boolean (default=‘‘True‘‘)

Whether or not to copy data when fitting and applying the model. Running the model in-place (`copy_data=False`) will be faster and have a smaller memory footprint, but if not done carefully can lead to data integrity issues.

Attributes

model	(A Scikit-learn pipeline (or equivalent)) The actual model used to compute WP. Upon initialization it will be set to a default model, but can be overridden by the user.
column_descriptions	(dictionary) A dictionary whose keys are the names of the columns used in the model, and the values are string descriptions of what the columns mean. Set at initialization to be the default model, if you create your own model you'll need to update this attribute manually.
training_seasons	(A list of ints, or `None` (default=''None'')) If the model was trained using data downloaded from nfldb, a list of the seasons used to train the model. If nfldb was **not** used, an empty list. If no model has been trained yet, `None`.
training_season_types	(A list of strings or `None` (default=''None'')) Same as `training_seasons`, except for the portions of the seasons used in training the model ("Preseason", "Regular", and/or "Postseason").
validation_seasons	(same as `training_seasons`, but for validation data.)
validation_season_types	(same as `training_season_types`, but for validation data.)
sample_probabilities	(A numpy array of floats or `None` (default=''None'')) After the model has been validated, contains the sampled predicted probabilities used to compute the validation statistic.
predicted_win_percents	(A numpy array of floats or `None` (default=''None'')) After the model has been validated, contains the actual probabilities in the test set at each probability in `sample_probabilities`.
num_plays_used	(A numpy array of floats or `None` (default=''None'')) After the model has been validated, contains the number of plays used to compute each element of `predicted_win_percents`.
model_directory	(string) The directory where all models will be saved to or loaded from.

create_default_pipeline()

Create the default win probability estimation pipeline.

Returns Scikit-learn pipeline

The default pipeline, suitable for computing win probabilities but by no means the best possible model.

This can be run any time a new default pipeline is required,

and either set to the `model` attribute or used independently.

classmethod load_model(*filename=None*)

Load a saved WPModel.

Parameters **Same as ''save_model''.**

Returns `nflwin.WPModel` instance.

model_directory = '/home/docs/checkouts/readthedocs.org/user_builds/nflwin/checkouts/1.0.0/nflwin/models'

num_plays_used

plot_validation(*axis=None*, ***kwargs*)

Plot the validation data.

Parameters **axis** : matplotlib.pyplot.axis object or `None` (default=''None'')

If provided, the validation line will be overlaid on `axis`. Otherwise, a new figure and axis will be generated and plotted on.

****kwargs**

Arguments to `axis.plot`.

Returns matplotlib.pylot.axis

The axis the plot was made on.

Raises **NotFittedError**

If the model hasn't been fit **and** validated.

predict_wp(*plays*)

Estimate the win probability for a set of plays.

Basically a simple wrapper around `WPModel.model.predict_proba`, takes in a DataFrame and then spits out an array of predicted win probabilities.

Parameters **plays** : Pandas DataFrame

The input data to use to make the predictions.

Returns Numpy array, of length `len(plays)`

Predicted probability that the offensive team in each play will go on to win the game.

Raises **NotFittedError**

If the model hasn't been fit.

predicted_win_percents

sample_probabilities

save_model(*filename=None*)

Save the WPModel instance to disk.

All models are saved to the same place, with the installed NFLWin library (given by `WPModel.model_directory`).

Parameters **filename** : string (default=None):

The filename to use for the saved model. If this parameter is not specified, save to the default filename. Note that if a model already lists with this filename, it will be overwritten. Note also that this is a filename only, **not** a full path. If a full path is specified it is likely (albeit not guaranteed) to cause errors.

Returns `None`

train_model(*source_data='nfldb', training_seasons=[2009, 2010, 2011, 2012, 2013, 2014], training_season_types=['Regular', 'Postseason'], target_colname='offense_won'*)

Train the model.

Once a modeling pipeline is set up (either the default or something custom-generated), historical data needs to be fed into it in order to "fit" the model so that it can then be used to predict future results. This method implements a simple wrapper around the core Scikit-learn functionality which does this.

The default is to use data from the nfldb database, however that can be changed to a simple Pandas DataFrame if desired (for instance if you wish to use data from another source).

There is no particular output from this function, rather the parameters governing the fit of the model are saved inside the model object itself. If you want to get an estimate of the quality of the fit, use the `validate_model` method after running this method.

Parameters **source_data** : the string `"nfldb"` or a Pandas DataFrame (default=""nfldb"")

The data to be used to train the model. If `"nfldb"`, will query the nfldb database for the training data (note that this requires a correctly configured installation of nfldb's database).

training_seasons : list of ints (default="[2009, 2010, 2011, 2012, 2013, 2014]")

What seasons to use to train the model if getting data from the nfldb database. If `source_data` is not `"nfldb"`, this argument will be ignored. **NOTE:** it is critical not to use all possible data in order to train the model - some will need to be reserved for a final validation (see the `validate_model` method). A good dataset to reserve for validation is the most recent one or two NFL seasons.

training_season_types : list of strings (default=‘‘[”Regular”, “Postseason”]‘‘)

If querying from the nfldb database, what parts of the seasons to use. Options are “Preseason”, “Regular”, and “Postseason”. If `source_data` is not `"nfldb"`, this argument will be ignored.

target_colname : string or integer (default=‘‘”offense_won”‘‘)

The name of the target variable column.

Returns `None`

Notes

If you are loading in the default model, **there is no need to re-run this method**. In fact, doing so will likely result in weird errors and could corrupt the model if you were to try to save it back to disk.

training_seasons

training_seasons_types

validate_model(*source_data='nfldb', validation_seasons=[2015], validation_season_types=['Regular', 'Postseason'], target_colname='offense_won'*)

Validate the model.

Once a modeling pipeline is trained, a different dataset must be fed into the trained model to validate the quality of the fit. This method implements a simple wrapper around the core Scikit-learn functionality which does this.

The default is to use data from the nfldb database, however that can be changed to a simple Pandas DataFrame if desired (for instance if you wish to use data from another source).

The output of this method is a p value which represents the confidence at which we can reject the null hypothesis that the model predicts the appropriate win probabilities. This number is computed by first smoothing the predicted win probabilities of both all test data and just the data where the offense won with a gaussian kernel density estimate with standard deviation = 0.01. Once the data is smooth, ratios at each percentage point from 1% to 99% are computed (i.e. what fraction of the time did the offense win when the model says they have a 1% chance of winning, 2% chance, etc.). Each of these ratios should be well approximated by the binomial distribution, since they are essentially independent (not perfectly but hopefully close enough) weighted coin flips, giving a p value. From there Fisher's method is used to combine the p values into a global p value. A p value close to zero means that the model is unlikely to be properly predicting the correct win probabilities. A p value close to one, **while not proof that the model is correct**, means that the model is at least not inconsistent with the hypothesis that it predicts good win probabilities.

Parameters **source_data** : the string `"nfldb"` or a Pandas DataFrame (default=‘‘”nfldb”‘‘)

The data to be used to train the model. If `"nfldb"`, will query the nfldb database for the training data (note that this requires a correctly configured installation of nfldb's database).

training_seasons : list of ints (default=‘‘[2015]‘‘)

What seasons to use to validate the model if getting data from the nfldb database. If `source_data` is not `"nfldb"`, this argument will be ignored. **NOTE:** it is critical

not to use the same data to validate the model as was used in the fit. Generally a good data set to use for validation is one from a time period more recent than was used to train the model. For instance, if the model was trained on data from 2009-2014, data from the 2015 season would be a sensible choice to validate the model.

training_season_types : list of strings (default=‘‘[”Regular”, “Postseason”]‘‘)

If querying from the nfldb database, what parts of the seasons to use. Options are “Preseason”, “Regular”, and “Postseason”. If `source_data` is not `"nfldb"`, this argument will be ignored.

target_colname : string or integer (default=‘‘”offense_won”‘‘)

The name of the target variable column.

Returns float, between 0 and 1

The combined p value, where smaller values indicate that the model is not accurately predicting win probabilities.

Raises **NotFittedError**

If the model hasn’t been fit.

Notes

Probabilities are computed between 1 and 99 percent because a single incorrect prediction at 100% or 0% automatically drives the global p value to zero. Since the model is being smoothed this situation can occur even when there are no model predictions at those extreme values, and therefore leads to erroneous p values.

While it seems reasonable (to me at least), I am not totally certain that this approach is entirely correct. It’s certainly sub-optimal in that you would ideally reject the null hypothesis that the model predictions **aren’t** appropriate, but that seems to be a much harder problem (and one that would need much more test data to beat down the uncertainties involved). I’m also not sure if using Fisher’s method is appropriate here, and I wonder if it might be necessary to Monte Carlo this. I would welcome input from others on better ways to do this.

validation_seasons

validation_seasons_types

nflwin.preprocessing module

Tools to get raw data ready for modeling.

class `nflwin.preprocessing.`**`CheckColumnNames`**(*column_names=None*, *copy=True*)

Bases: `sklearn.base.BaseEstimator`

Make sure user has the right column names, in the right order.

This is a useful first step to make sure that nothing is going to break downstream, but can also be used effectively to drop columns that are no longer necessary.

Parameters **column_names** : `None`, or list of strings

A list of column names that need to be present in the scoring data. All other columns will be stripped out. The order of the columns will be applied to any scoring data as well, in order to handle the fact that pandas lets you play fast and loose with column order. If `None`, will obtain every column in the DataFrame passed to the `fit` method.

copy : boolean (default=‘‘True‘‘)

If `False`, add the score differential in place.

fit(*X*, *y=None*)

Grab the column names from a Pandas DataFrame.

Parameters **X** : Pandas DataFrame, of shape(number of plays, number of features)

NFL play data.

y : Numpy array, with length = number of plays, or None

1 if the home team won, 0 if not. (Used as part of Scikit-learn's `Pipeline`)

Returns **self** : For compatibility with Scikit-learn's `Pipeline`.

transform(*X*, *y=None*)

Apply the column ordering to the data.

Parameters **X** : Pandas DataFrame, of shape(number of plays, number of features)

NFL play data.

y : Numpy array, with length = number of plays, or None

1 if the home team won, 0 if not. (Used as part of Scikit-learn's `Pipeline`)

Returns **X** : Pandas DataFrame, of shape(number of plays, `len(column_names)`)

The input DataFrame, properly ordered and with extraneous columns dropped

Raises **KeyError**

If the input data frame doesn't have all the columns specified by `column_names`.

NotFittedError

If `transform` is called before `fit`.

class `nflwin.preprocessing.`**ComputeElapsedTime**(*quarter_colname*, *quarter_time_colname*, *quarter_to_second_mapping={'Q1': 0, 'Q3': 1800, 'Q2': 900, 'Q4': 2700, 'OT3': 5400, 'OT2': 4500, 'OT': 3600}*, *total_time_colname='total_elapsed_time'*, *copy=True*)

Bases: `sklearn.base.BaseEstimator`

Compute the total elapsed time from the start of the game.

Parameters **quarter_colname** : string

Which column indicates what quarter it is.

quarter_time_colname : string

Which column indicates how much time has elapsed in the current quarter.

quarter_to_second_mapping : dict (default=‘‘{“Q1”: 0, “Q2”: 900, “Q3”: 1800, “Q4”: 2700,

“OT”: 3600, “OT2”: 4500, “OT3”: 5400}‘‘)

What mapping to use between the string values in the quarter column and the seconds they correspond to. Mostly useful if your data had quarters listed as something like “Quarter 1” or “q1” instead of the values from `nfldb`.

total_time_colname : string (default=”total_elapsed_time”)

What column name to store the total elapsed time under.

copy : boolean (default=True)

Whether to add the new column in place.

fit(*X*, *y=None*)

transform(*X*, *y=None*)

Create the new column.

Parameters **X** : Pandas DataFrame, of shape(number of plays, number of features)

NFL play data.

y : Numpy array, with length = number of plays, or None

1 if the home team won, 0 if not. (Used as part of Scikit-learn's `Pipeline`)

Returns **X** : Pandas DataFrame, of shape(number of plays, number of features + 1)

The input DataFrame, with the new column added.

Raises **KeyError**

If `quarter_colname` or `quarter_time_colname` don't exist, or if `total_time_colname` **does** exist.

TypeError

If the total time elapsed is not a numeric column, which typically indicates that the mapping did not apply to every row.

class `nflwin.preprocessing.`**ComputeIfOffenseIsHome**(*offense_team_colname*, *home_team_colname*, *offense_home_team_colname='is_offense_home'*, *copy=True*)

Bases: `sklearn.base.BaseEstimator`

Determine if the team currently with possession is the home team.

Parameters **offense_team_colname** : string

Which column indicates what team was on offense.

home_team_colname : string

Which column indicates what team was the home team.

offense_home_team_colname : string (default="is_offense_home")

What column to store whether or not the offense was the home team.

copy : boolean (default=True)

Whether to add the new column in place.

fit(*X*, *y=None*)

transform(*X*, *y=None*)

Create the new column.

Parameters **X** : Pandas DataFrame, of shape(number of plays, number of features)

NFL play data.

y : Numpy array, with length = number of plays, or None

1 if the home team won, 0 if not. (Used as part of Scikit-learn's `Pipeline`)

Returns **X** : Pandas DataFrame, of shape(number of plays, number of features + 1)

The input DataFrame, with the new column added.

Raises **KeyError**

If `offense_team_colname` or `home_team_colname` don't exist, or if `offense_home_team_colname` **does** exist.

class `nflwin.preprocessing.`**`CreateScoreDifferential`**(*home_score_colname*, *away_score_colname*, *offense_home_colname*, *score_differential_colname='score_differential'*, *copy=True*)

Bases: `sklearn.base.BaseEstimator`

Convert offense and defense scores into a differential (offense - defense).

Parameters **home_score_colname** : string

The name of the column containing the score of the home team.

away_score_colname : string

The name of the column containing the score of the away team.

offense_home_colname : string

The name of the column indicating if the offense is home.

score_differential_colname : string (default=‘‘”score_differential”‘‘)

The name of column containing the score differential. Must not already exist in the DataFrame.

copy : boolean (default = `True`)

If `False`, add the score differential in place.

`fit`(*X*, *y=None*)

`transform`(*X*, *y=None*)

Create the score differential column.

Parameters **X** : Pandas DataFrame, of shape(number of plays, number of features)

NFL play data.

y : Numpy array, with length = number of plays, or None

1 if the home team won, 0 if not. (Used as part of Scikit-learn's `Pipeline`)

Returns **X** : Pandas DataFrame, of shape(number of plays, number of features + 1)

The input DataFrame, with the score differential column added.

class `nflwin.preprocessing.`**`MapToInt`**(*colname*, *copy=True*)

Bases: `sklearn.base.BaseEstimator`

Map a column of values to integers.

Mapping to integer is nice if you know a column only has a few specific values in it, but you need to convert it to integers before one-hot encoding it.

Parameters **colname** : string

The name of the column to perform the mapping on.

copy : boolean (default=True)

If `False`, apply the mapping in-place.

Attributes

mapping	(dict) Keys are the unique values of the column, values are the integers those values will be mapped to.

fit(*X*, *y=None*)

Find all unique strings and construct the mapping.

Parameters **X** : Pandas DataFrame, of shape(number of plays, number of features)

NFL play data.

y : Numpy array, with length = number of plays, or None

1 if the home team won, 0 if not. (Used as part of Scikit-learn's `Pipeline`)

Returns **self** : For compatibility with Scikit-learn's `Pipeline`.

Raises **KeyError**

If `colname` is not in `X`.

transform(*X*, *y=None*)

Apply the mapping to the data.

Parameters **X** : Pandas DataFrame, of shape(number of plays, number of features)

NFL play data.

y : Numpy array, with length = number of plays, or None

1 if the home team won, 0 if not. (Used as part of Scikit-learn's `Pipeline`)

Returns **X** : Pandas DataFrame, of shape(number of plays, number of features)

The input DataFrame, with the mapping applied.

Raises **NotFittedError**

If `transform` is called before `fit`.

KeyError

If `colname` is not in `X`.

class nflwin.preprocessing.**OneHotEncoderFromDataFrame**(*categorical_feature_names='all'*, *dtype=<type 'float'>*, *handle_unknown='error'*, *copy=True*)

Bases: `sklearn.base.BaseEstimator`

One-hot encode a DataFrame.

This cleaner wraps the standard scikit-learn OneHotEncoder, handling the transfer between column name and column index.

Parameters **categorical_feature_names** : "all" or array of column names.

Specify what features are treated as categorical. * "all" (default): All features are treated as categorical. * array of column names: Array of categorical feature names.

dtype : number type, default=np.float.

Desired dtype of output.

handle_unknown : str, "error" (default) or "ignore".

Whether to raise an error or ignore if an unknown categorical feature is present during transform.

copy : boolean (default=True)

If `False`, apply the encoding in-place.

dtype

fit(*X*, *y=None*)

Convert the column names to indices, then compute the one hot encoding.

Parameters **X** : Pandas DataFrame, of shape(number of plays, number of features)

NFL play data.

y : Numpy array, with length = number of plays, or None

1 if the home team won, 0 if not. (Used as part of Scikit-learn's `Pipeline`)

Returns **self** : For compatibility with Scikit-learn's `Pipeline`.

handle_unknown

transform(*X*, *y=None*)

Apply the encoding to the data.

Parameters **X** : Pandas DataFrame, of shape(number of plays, number of features)

NFL play data.

y : Numpy array, with length = number of plays, or None

1 if the home team won, 0 if not. (Used as part of Scikit-learn's `Pipeline`)

Returns **X** : Pandas DataFrame, of shape(number of plays, number of new features)

The input DataFrame, with the encoding applied.

nflwin.utilities module

Utility functions that don't fit in the main modules

nflwin.utilities.**connect_nfldb**()

Connect to the nfldb database.

Rather than using the builtin method we make our own, since we're going to use SQLAlchemy as the engine. However, we can still make use of the information in the nfldb config file to get information like username and password, which means this function doesn't need any arguments.

Parameters **None**

Returns SQLAlchemy engine object

A connected engine, ready to be used to query the DB.

Raises **IOError**

If it can't find the config file.

nflwin.utilities.**get_nfldb_play_data**(*season_years=None, season_types=['Regular', 'Post-season']*)

Get play-by-play data from the nfldb database.

We use a specialized query and then postprocessing because, while possible to do using the objects created by `nfldb`, it is *orders of magnitude slower*. This is due to the more general nature of `nfldb`, which is not really

designed for this kind of data mining. Since we need to get a lot of data in a single way, it's much simpler to interact at a lower level with the underlying postgres database.

Parameters **season_years** : list (default=None)

A list of all years to get data for (earliest year in nfldb is 2009). If `None`, get data from all available seasons.

season_types : list (default=["Regular", "Postseason"])

A list of all parts of seasons to get data for (acceptable values are "Preseason", "Regular", and "Postseason"). If `None`, get data from all three season types.

Returns Pandas DataFrame

The play by play data, with the following columns:

- **gsis_id:** The official NFL GSIS_ID for the game.
- **drive_id:** The id of the drive, starts at 1 and increases by 1 for each new drive.
- **play_id:** The id of the play in `nfldb`. Note that sequential plays have increasing but not necessarily sequential values. With `drive_id` and `gsis_id`, works as a unique identifier for a given play.
- **quarter:** The quarter, prepended with "Q" (e.g. `Q1` means the first quarter). Overtime periods are denoted as `OT`, `OT2`, and theoretically `OT3` if one were to ever be played.
- **seconds_elapsed:** seconds elapsed since the start of the quarter.
- **offense_team:** The abbreviation of the team currently with possession of the ball.
- **yardline:** The current field position. Goes from -49 to 49, where negative numbers indicate that the team with possession is on its own side of the field.
- **down:** The down. kickoffs, extra points, and similar have a down of 0.
- **yards_to_go:** How many yards needed in order to get a first down (or touchdown).
- **home_team:** The abbreviation of the home team.
- **away_team:** The abbreviation of the away team.
- **curr_home_score:** The home team's score at the start of the play.
- **curr_away_score:** The away team's score at the start of the play.
- **offense_won:** A boolean - `True` if the offense won the game, `False` otherwise. (The database query skips tied games.)

Notes

`gsis_id`, `drive_id`, and `play_id` are not necessary to make the model, but are included because they can be useful for computing things like WPA.

Module contents

n

nflwin, 27
nflwin.model, 17
nflwin.preprocessing, 21
nflwin.utilities, 26

C

CheckColumnNames (class in nflwin.preprocessing), 21
ComputeElapsedTime (class in nflwin.preprocessing), 22
ComputeIfOffenseIsHome (class in nflwin.preprocessing), 23
connect_nfldb() (in module nflwin.utilities), 26
create_default_pipeline() (nflwin.model.WPModel method), 18
CreateScoreDifferential (class in nflwin.preprocessing), 24

D

dtype (nflwin.preprocessing.OneHotEncoderFromDataFrame attribute), 26

F

fit() (nflwin.preprocessing.CheckColumnNames method), 22
fit() (nflwin.preprocessing.ComputeElapsedTime method), 23
fit() (nflwin.preprocessing.ComputeIfOffenseIsHome method), 23
fit() (nflwin.preprocessing.CreateScoreDifferential method), 24
fit() (nflwin.preprocessing.MapToInt method), 25
fit() (nflwin.preprocessing.OneHotEncoderFromDataFrame method), 26

G

get_nfldb_play_data() (in module nflwin.utilities), 26

H

handle_unknown (nflwin.preprocessing.OneHotEncoderFromDataFrame attribute), 26

L

load_model() (nflwin.model.WPModel class method), 18

M

MapToInt (class in nflwin.preprocessing), 24
model_directory (nflwin.model.WPModel attribute), 18

N

nflwin (module), 27
nflwin.model (module), 17
nflwin.preprocessing (module), 21
nflwin.utilities (module), 26
num_plays_used (nflwin.model.WPModel attribute), 18

O

OneHotEncoderFromDataFrame (class in nflwin.preprocessing), 25

P

plot_validation() (nflwin.model.WPModel method), 18
predict_wp() (nflwin.model.WPModel method), 19
predicted_win_percents (nflwin.model.WPModel attribute), 19

S

sample_probabilities (nflwin.model.WPModel attribute), 19
save_model() (nflwin.model.WPModel method), 19

T

train_model() (nflwin.model.WPModel method), 19
training_seasons (nflwin.model.WPModel attribute), 20
training_seasons_types (nflwin.model.WPModel attribute), 20
transform() (nflwin.preprocessing.CheckColumnNames method), 22
transform() (nflwin.preprocessing.ComputeElapsedTime method), 23
transform() (nflwin.preprocessing.ComputeIfOffenseIsHome method), 23
transform() (nflwin.preprocessing.CreateScoreDifferential method), 24
transform() (nflwin.preprocessing.MapToInt method), 25
transform() (nflwin.preprocessing.OneHotEncoderFromDataFrame method), 26

V

validate_model() (nflwin.model.WPModel method), 20
validation_seasons (nflwin.model.WPModel attribute), 21
validation_seasons_types (nflwin.model.WPModel attribute), 21

W

WPModel (class in nflwin.model), 17